DevKit

Release 1.2

Digital Substrate

Apr 30, 2026

DSM - DATA MODELING

Define your data model. Get type safety, versioning, and Python bindings for free.

DevKit is the Digital Substrate Python toolkit for metadata-driven data modeling. Define your data structures in DSM (a purpose-built DSL), and the dsviper runtime gives you:

- **Strong typing** - Type mismatches raise exceptions immediately, not silently corrupt data
- **Version control for data** - Every mutation is tracked in a commit DAG (like git for your data)
- **Seamless Python integration** - Work with native Python types, dsviper handles conversions

```
from dsviper import CommitDatabase, CommitMutableState
import model.attachments as ma

# Open a versioned database
db = CommitDatabase.open("model.cdb")

# Create and modify typed data
key = ma.Tuto_UserKey.create()
login = ma.Tuto_Login()
login.nickname = "alice"

# Commit with full history
state = CommitMutableState(db.initial_state())
ma.tuto_user_login_set(state.attachment_mutating(), key, login)
db.commit_mutations("Add user", state)
```

CHAPTER

ONE

DEVKIT, DSVIPER, VIPER — WHAT'S WHAT?

Three names appear across this documentation. Here is how they layer.

DevKit — *What you download.*
: The Digital Substrate Python toolkit, distributed as a single ZIP — bundles dsviper, the Kibo code generator, templates, CLI tools, and offline documentation.

dsviper — *What runs in your Python apps.*
: The Python runtime distributed on PyPI. Strong-typed Python API over the Viper C++ engine. Installed via `pip install dsviper` or pulled in by the DevKit ZIP.

Viper — *The C++ engine behind dsviper.*
: The underlying metadata-driven C++ runtime — type system, commit DAG, database tier. Currently used internally; not yet distributed standalone.

CHAPTER

TWO

TWO WAYS TO WORK

Static API - Generate type-safe Python packages from your DSM definitions. Get IDE autocompletion, type checking, and domain-specific APIs.

Dynamic API - Use Viper's runtime metadata directly. No code generation needed. Load definitions at runtime, introspect types, build tools that work with any schema.

Both approaches use the same Viper runtime. Choose based on your needs.

CHAPTER

THREE

DOCUMENTATION

3.1 DSM - Data Modeling Manual

The **Digital Substrate Model (DSM)** is a domain-specific language for defining data models. This manual covers the DSM language, its type system, and how it integrates with code generation and application architecture.

3.1.1 Introduction to DSM

The **Digital Substrate Model (DSM)** is a domain-specific language (DSL) for defining data models. DSM enables you to define aggregates of information linked by unique, strongly-typed keys to construct complex structured data.

The Five Fundamental Notions

To define a structured and flexible data model, DSM introduces five fundamental notions:

Notion	Description	Example
Namespace	A space where types are defined, identified by a UUID	`namespace Graph {27c49329-...}`
Concept	An abstract thing (like an abstract type)	`concept Vertex;`
Key	A way to identify an instance of a Concept	`key<Vertex>`
Document	A piece of information expressible in the type system	`struct Login { ... }`
Attachment	A way to associate a Document with a Key	`attachment<User, Login> login;`

Namespace

A namespace groups related definitions and provides isolation. Each namespace has a unique UUID that serves as the seed to generate RuntimeIds for all types defined within.

```
namespace Graph {27c49329-a399-415c-baf0-db42949d2ba2} {
    // All definitions here belong to the Graph namespace
};
```

Concept

A **concept** is an abstract term from your data domain. It represents something that exists only through its instances (keys). Concepts can form hierarchies using `is a`.

```
// From Graph Editor: three independent concepts
concept Graph;
concept Vertex;
concept Edge;
```

Concepts can have inheritance relationships:

```
// From Raptor Editor: Material hierarchy
concept Material;
concept MaterialStandard is a Material;
concept MaterialMirror is a Material;
concept MaterialMatte is a Material;
```

Key

A **key** identifies a specific instance of a concept. Think of it as a typed UUID: `key<Vertex>` can only reference a Vertex, not an Edge.

Keys are created at runtime and persist across sessions:

```
# Python example
vertex_key = Graph_VertexKey.create()
print(vertex_key.instance_id())  # fc472756-8f9e-42aa-a06f-051d330d0108
```

Document

A **document** is any data expressible in the DSM type system. The most common form is a structure:

```
struct Login {
    string nickname;
    string password;
};

struct Position {
    float x;
    float y;
};
```

Attachment

An **attachment** associates a document type with a key type. It defines what data can be stored for each instance of a concept.

```
// Each User can have a Login document
attachment<User, Login> login;

// Each Vertex can have position data
attachment<Vertex, Vertex2DAttributes> render2DAttributes;
```

Data Definitions Extracted from Code

DSM extracts data definitions from the programming language. Instead of scattering struct definitions across C++ headers, DSM captures the data model in a single, language-independent notation — enabling Kibo to generate infrastructure for multiple targets from one source of truth.

This means DSM can adapt to an existing codebase: translating C++ data structures into DSM is a straightforward extraction of what already exists. The *Raptor Editor sample* demonstrates this approach, where an industrial C++ data model was translated directly into DSM.

The Data Model is a Set of Sealed Definitions

Unlike traditional databases where schemas have versions, DSM takes a different approach:

- Each definition is **sealed by its definition** in a namespace
- The **RuntimeId** is computed deterministically from the definition
- There is **no schema version** - the data model is a set of immutable definitions

This enables schema evolution through adding new attachments rather than migrating existing data (see *Attachments* for details).

Example: A Complete Data Model

Here is a complete data model from the Graph Editor, demonstrating the **recommended pattern** with multiple attachments per concept:

```
namespace Graph {27c49329-a399-415c-baf0-db42949d2ba2} {

// Concepts
concept Graph;
concept Vertex;
concept Edge;

// Documents for Graph
struct GraphTopology {
    set<key<Vertex>> vertexKeys;
    set<key<Edge>> edgeKeys;
};

struct GraphSelection {
    set<key<Vertex>> vertexKeys;
    set<key<Edge>> edgeKeys;
};

struct GraphDescription {
    string name;
    string author;
    string createDate;
};

// Documents for Vertex
struct Position {
    float x;
    float y;
};

struct Color {
    float red;
    float green;
    float blue;
};

struct Vertex2DAttributes {
    Position position;
```

(continues on next page)

(continued from previous page)

```
};

struct VertexVisualAttributes {
    int64 value;
    Color color;
};

// Document for Edge
struct EdgeTopology {
    key<Vertex> vaKey;
    key<Vertex> vbKey;
};

// Graph attachments - each concern is separate
attachment<Graph, GraphTopology> topology;          // Structure
attachment<Graph, GraphSelection> selection;        // UI state
attachment<Graph, GraphDescription> description;    // Metadata
attachment<Graph, map<string, string>> tags;        // User-defined tags
attachment<Graph, xarray<string>> comments;         // Ordered comments

// Vertex attachments - rendering and appearance are separate
attachment<Vertex, Vertex2DAttributes> render2DAttributes;
attachment<Vertex, VertexVisualAttributes> visualAttributes;

// Edge attachment
attachment<Edge, EdgeTopology> topology;

};
```

Why multiple attachments matter:

- A Vertex can exist with only position data (no visual attributes)
- Selection state is separate from topology (UI concern vs data concern)
- Adding a new feature (e.g., `physicsAttributes`) requires **no migration**
- Each attachment is independent and optional

This model defines:

- Three concepts (Graph, Vertex, Edge)
- Eight document types (structures for different concerns)
- Eight attachments (five for Graph, two for Vertex, one for Edge)

What's Next

- *Types and Structures* - Learn about all available types in DSM
- *Concepts and Hierarchies* - Understand concept inheritance with `is a`
- *Attachments* - Master the recommended patterns for schema design

3.1.2 Types and Structures

DSM provides a rich type system for modeling data. This chapter covers all available types, from primitives to complex generic containers.

Primitive Types

Boolean

The `bool` type represents true/false values.

```
struct CameraDepthOfField {
    bool enabled;
    float aperture;
    int32 sampleCount = 32;
};
```

Integers

DSM supports both signed and unsigned integers of various sizes:

Type	Range	C++ Mapping
`uint8`	0 to 255	`std::uint8_t`
`uint16`	0 to 65,535	`std::uint16_t`
`uint32`	0 to 4 billion	`std::uint32_t`
`uint64`	0 to 18 quintillion	`std::uint64_t`
`int8`	-128 to 127	`std::int8_t`
`int16`	-32,768 to 32,767	`std::int16_t`
`int32`	-2 billion to 2 billion	`std::int32_t`
`int64`	±9 quintillion	`std::int64_t`

```
// From Raptor Editor: Iray rendering settings
struct IraySettingsProperties {
    uint32 maxSamples = 512;
    uint32 maxPathLength = 12;
    float maxRenderTime = 3600.0;  // In seconds
};
```

Real Numbers

The `float` and `double` types represent floating-point numbers.

```
// From Raptor Editor: Camera optical properties
struct CameraOpticalProperties {
    float fov = 0.785398;  // Field of view in radians
    CameraDepthOfField depthOfField;
    CameraMotionBlur motionBlur;
};
```

String

The `string` type stores UTF-8 encoded text.

```
struct Login {
    string nickname;
    string password;
};
```

UUID

The `uuid` type stores universally unique identifiers (UUID4).

```
struct S {
    uuid identifier;
    uuid reference = {8f2586fc-735b-48ca-8d32-3b7545f65cd6};
};
```

Blob and BlobId

DSM provides two types for binary data:

- `blob`: Inline binary data (for small payloads like thumbnails)
- `blob_id`: Reference to external binary data (for large payloads like textures)

```
// From Raptor Editor: Mesh geometry data
struct MeshProperties {
    Aabb aabb;
    int32 triangleCount;
    int32 vertexCount;
    blob_id blob_indices;
    blob_id blob_positions;
    blob_id blob_normals;
    blob_id blob_tangents;
};

// Small data inline, large data by reference
struct Image {
    uint16 width;
    uint16 height;
    blob thumbnail;     // Small: 32x32 preview
    blob_id pixels;     // Large: full resolution
};
```

Enumerations

The `enum` type defines a fixed set of named values. Enumerations are limited to 256 cases.

```
// From Raptor Editor: Material types
enum MaterialStandardType {
    diffuse,
    diffuseSpecular,
    transparent
```

(continues on next page)

(continued from previous page)

```
};

// From Raptor Editor: Light attenuation models
enum LightAttenuationType {
    none,
    linearSlow,
    linearFast,
    quadraticSlow,
    quadraticFast,
    physical
};
```

Use the dot prefix to reference enum values in field initializers:

```
struct MaterialStandardProperties {
    string name = "MaterialStandard";
    MaterialStandardType materialType = .diffuseSpecular;
    // ...
};
```

Structures

Structures aggregate fields into composite types. Fields can have default values.

```
// From Raptor Editor: Basic 3D vector
struct Vector {
    float x;
    float y;
    float z;
};

// From Raptor Editor: Axis-aligned bounding box
struct Aabb {
    Vector min;
    Vector max;
};

// From Raptor Editor: 3D transformation
struct Transform {
    Vector translation;
    Vector orientation;
    Vector scaling = {1.0, 1.0, 1.0};  // Default: no scaling
};
```

Structures can be nested but **cannot be recursive**. For recursive relationships, use `optional<key<T>>`:

```
// From Raptor Editor: Configuration expression tree
struct ConfigurationExpressionProperties {
    ConfigurationExpressionOperationType operationType = .defined;
    optional<key<ConfigurationExpression>> leftExpressionKey;   // Recursive via key
    optional<key<ConfigurationExpression>> rightExpressionKey;  // Recursive via key
    string symbol;
};
```

Mathematical Types

Vec

The `vec<T, n>` type creates fixed-size numeric arrays, useful for graphics:

```
struct Mesh {
    vector<vec<float, 3>> positions;  // XYZ coordinates
    vector<vec<float, 2>> uvs;        // UV texture coordinates
};
```

Mat

The `mat<T, columns, rows>` type creates matrices:

```
struct S {
    mat<float, 4, 4> transform;  // 4x4 transformation matrix
};
```

Generic Types

Vector

The `vector<T>` type is a dynamic array.

```
// From Raptor Editor: Camera groups contain cameras
struct CameraGroupProperties {
    string name = "Camera Group";
    vector<key<CameraGroup>> groupKeys;  // Child groups
    vector<key<Camera>> cameraKeys;      // Cameras in this group
};

// From Raptor Editor: Bezier path with control points
struct BezierPathProperties {
    string name = "BezierPath";
    Vector color = {1.0, 1.0, 1.0};
    vector<BezierPathVertex> vertices;
};
```

Set

The `set<T>` type is an unordered collection of unique values.

```
// From Raptor Editor: Surface tags for filtering
struct SurfaceProperties {
    string name = "Surface";
    set<string> tags;
    // ...
};

// From Raptor Editor: Configuration defines
struct ProductProperties {
    // ...
```

(continues on next page)

(continued from previous page)

```
    set<string> configurationDefines;
    map<string, set<string>> configurationBookmarks;
};
```

Map

The `map<K, V>` type associates keys with values.

```
// From Raptor Editor: Material assignments per surface
struct AspectLayerProperties {
    string name = "AspectLayer";
    bool enabled;
    map<key<Surface>, MaterialAssignment> materialAssignments;
    map<key<Surface>, vector<LabelAssignment>> labelAssignments;
};

// From Raptor Editor: Environment assignments
struct EnvironmentLayerProperties {
    string name = "EnvironmentLayer";
    bool enabled = true;
    map<key<Surface>, key<Environment>> environmentAssignments;
    map<key<Surface>, Transform> orientationAssignments;
};
```

Optional

The `optional<T>` type represents a value that may or may not be present.

```
// From Raptor Editor: Optional references
struct CameraProperties {
    string name = "Camera";
    PointOfView pointOfView;
    CameraOpticalProperties opticalProperties;
    optional<key<Sensor>> sensorKey;  // May not have a sensor
};

struct MaterialStandardProperties {
    // ...
    optional<key<Texture>> diffuseMapKey;
    optional<key<BumpMap>> bumpMapKey;
    // ...
};
```

Tuple

The `tuple<T0, ...>` type groups heterogeneous values.

```
struct S {
    tuple<float, float> location;
    tuple<string, int32, bool> metadata;
};
```

Variant

The `variant<T0, ...>` type holds one of several possible types.

```
// Multi-layer material with different layer types
struct MaterialMultilayer {
    vector<variant<LayerIllumination, LayerDiffuse, LayerSpecular>> layers;
};
```

XArray

The `xarray<T>` type is like a vector but preserves element order during concurrent mutations. Designed for multiplayer editing scenarios.

```
struct Document {
    xarray<string> comments;  // Order preserved across concurrent edits
};
```

Special Types

Key

The `key<T>` type references an instance of a concept. It acts like a strongly-typed UUID:

```
// From Raptor Editor: Model references various concepts
struct ModelProperties {
    string name = "Model";
    vector<key<LightingLayer>> lightingLayerKeys;
    key<GeometryLayer> rootGeometryLayerKey;
    key<Kinematics> kinematicsKey;
    vector<key<PositionLayer>> positionLayerKeys;
};
```

Keys enable:

- **Type safety**: `key<Vertex>` cannot accidentally reference an `Edge`
- **Polymorphism**: `key<Material>` can reference any material subtype

Any

The `any` type holds any value, requiring runtime type checking:

```
// Store arbitrary documents for a user
attachment<User, any> documents;
```

Use `any` sparingly—prefer strongly-typed alternatives when possible.

Default Values

Fields can specify default values. If not specified, fields are initialized to the type's "zero":

Type	Zero Value
`bool`	`false`
integers	`0`
`float, double`	`0.0`
`string`	`""` (empty string)
`uuid`	nil UUID
containers	empty
`optional`	empty (nil)
struct	all fields at zero

```
// From Raptor Editor: Transform with sensible defaults
struct Transform {
    Vector translation;              // {0, 0, 0}
    Vector orientation;              // {0, 0, 0}
    Vector scaling = {1.0, 1.0, 1.0}; // Identity scale
};

// From Raptor Editor: Camera view defaults
struct PointOfView {
    Vector target;                    // {0, 0, 0}
    Vector eye = {2.0, 2.0, 2.0};     // Offset from origin
    Vector up = {0.0, 1.0, 0.0};      // Y-up convention
};
```

What's Next

- *Concepts and Hierarchies* - Learn about concept inheritance
- *Attachments* - Associate documents with keys

3.1.3 Concepts and Hierarchies

Concepts are the abstract building blocks of your data model. They represent entities that exist only through their instances (keys).

Basic Concepts

A concept declares an abstract entity from your domain:

```
// From Graph Editor: Three independent concepts
concept Graph;
concept Vertex;
concept Edge;
```

These concepts have no concrete implementation—only their keys exist at runtime:

```
# Python: Creating concept instances
vertex_key = Graph_VertexKey.create()
print(vertex_key.instance_id())  # fc472756-8f9e-42aa-a06f-051d330d0108
```

Concept Inheritance

Concepts can form hierarchies using `is a`:

```
// From Raptor Editor: Material hierarchy
concept Material;
concept MaterialStandard is a Material;
concept MaterialMultilayer is a Material;
concept MaterialMirror is a Material;
concept MaterialMatte is a Material;
concept MaterialEnvironment is a Material;
concept MaterialSeam is a Material;
concept MaterialAxfCpa2 is a Material;
```

This hierarchy enables **polymorphism**: a `key<Material>` can reference any material subtype.

Light Hierarchy Example

```
// From Raptor Editor: Light types
concept Light;
concept LightSpot is a Light;
concept LightOmni is a Light;
concept LightSun is a Light;
concept LightSky is a Light;
concept LightAreaPlane is a Light;
concept LightAreaCylinder is a Light;
concept LightAreaMesh is a Light;
```

Each light type has its own properties structure:

```
// Base properties (all lights have these)
struct LightProperties {
    string name;
    bool enabled = true;
    Vector color = {1.0, 1.0, 1.0};
    float intensity = 1.0;
    Vector position = {0.0, 0.1, 0.0};
    Vector orientation;
};

// Spot-specific properties
struct LightSpotProperties {
    float diameter = 0.05;
    Vector target = {0.0, 0.0, 1.0};
    float falloff = 45.0;  // Degrees
    float hotSpot = 43.0;  // Degrees
    LightShadow shadow;
    LightAttenuation attenuation;
};

// Area light properties
struct LightAreaPlaneProperties {
    float width = 1.0;   // Meters
    float height = 1.0;  // Meters
```

(continues on next page)

(continued from previous page)

```
    LightShadow shadow;
    LightAttenuation attenuation;
};
```

Multi-level Hierarchies

Hierarchies can have multiple levels:

```
// From Raptor Editor: Environment generator hierarchy
concept EnvironmentGenerator;
concept EnvironmentGeneratorHdrls is a EnvironmentGenerator;
concept EnvironmentGeneratorLocal is a EnvironmentGenerator;

// From Raptor Editor: Kinematics node hierarchy
concept KinematicsNode;
concept KinematicsNodeAxis is a KinematicsNode;
concept KinematicsNodeNull is a KinematicsNode;
concept KinematicsNodeVector is a KinematicsNode;

// Kinematics constraint hierarchy
concept KinematicsConstraint;
concept KinematicsConstraintCopyOrientation is a KinematicsConstraint;
concept KinematicsConstraintCopyPosition is a KinematicsConstraint;
concept KinematicsConstraintFollowPath is a KinematicsConstraint;
concept KinematicsConstraintLookAt is a KinematicsConstraint;
```

Key Polymorphism

Keys respect the inheritance hierarchy:

```
// A structure that accepts any material
struct MaterialAssignment {
    key<Material> materialKey;  // Can hold any material subtype
    Transform transform;
    int8 uvSet;
};
```

At runtime, the generated code provides type-safe conversions between parent and child keys:

```
# Generated code from red/data.py

# Parent key can try to convert to a specific child (returns None if not that type)
light_key = Raptor_LightKey.create()
spot_key = light_key.to_raptor_light_spot_key()  # -> Raptor_LightSpotKey | None

# Child key can convert to parent key
spot_key = Raptor_LightSpotKey.create()
parent_key = spot_key.to_parent_key()  # -> Raptor_LightKey

# Parent key can be constructed from any child key
omni_key = Raptor_LightOmniKey.create()
light_key = Raptor_LightKey.from_raptor_light_omni_key(omni_key)  # -> Raptor_LightKey
```

Clubs: Grouping Unrelated Concepts

Sometimes you need to reference concepts that share no inheritance relationship. A `club` groups unrelated concepts:

```
// From Raptor Editor: Configuration target sources
// These concepts are unrelated but can all be configuration sources
club ConfigurationTargetSource;
membership ConfigurationTargetSource Model;
membership ConfigurationTargetSource Product;
membership ConfigurationTargetSource Overlay;

// From Raptor Editor: Configuration target elements
club ConfigurationTargetElement;
membership ConfigurationTargetElement GeometryLayer;
membership ConfigurationTargetElement AspectLayer;
membership ConfigurationTargetElement PositionLayer;
membership ConfigurationTargetElement EnvironmentLayer;
membership ConfigurationTargetElement LightingLayer;
membership ConfigurationTargetElement LightingLayerColorLayer;
membership ConfigurationTargetElement OverlayLayer;
```

Use club keys to reference any member:

```
// From Raptor Editor: Configuration target
struct ConfigurationTarget {
    string name = "ConfigurationTarget";
    bool enabled;
    key<ConfigurationTargetSource> sourceKey;    // Model, Product, or Overlay
    key<ConfigurationTargetElement> elementKey; // Any layer type
};
```

Kinematics Club Example

```
// From Raptor Editor: Elements that can be animated
club KinematicsElement;
membership KinematicsElement Surface;
membership KinematicsElement BezierPath;
membership KinematicsElement KinematicsNodeAxis;
membership KinematicsElement KinematicsNodeNull;
membership KinematicsElement KinematicsNodeVector;

// Kinematics uses this to track parent-child relationships
struct KinematicsProperties {
    vector<key<KinematicsNode>> nodeKeys;
    map<key<KinematicsElement>, vector<key<KinematicsConstraint>>> constraints;
    map<key<KinematicsElement>, key<KinematicsElement>> parents;
};
```

Concept vs Club

Aspect	Concept Hierarchy	Club
Relationship	"is a" (inheritance)	"member of" (grouping)
Use case	Related types with shared semantics	Unrelated types with shared usage
Example	`MaterialMirror is a Material`	`membership ConfigurationTargetSource Model`
Membership	Single parent	Multiple clubs allowed

A concept can be a member of multiple clubs:

RuntimeId Generation

Each concept receives a unique **RuntimeId** computed deterministically from its definition within its namespace. This means:

- The same concept definition always produces the same RuntimeId
- Different namespaces produce different RuntimeIds even for identically-named concepts
- RuntimeIds enable type-safe serialization and persistence

```
namespace Graph {27c49329-a399-415c-baf0-db42949d2ba2} {
concept Vertex;  // RuntimeId computed from namespace UUID + "Vertex"
};

namespace Other {12345678-1234-1234-1234-123456789abc} {
concept Vertex;  // Different RuntimeId (different namespace)
};
```

What's Next

- *Attachments* - Associate data with concept instances
- *Function Pools* - Define operations on your data model

3.1.4 Attachments

Attachments associate documents (data) with keys (concept instances). They are the primary mechanism for storing information about your domain entities.

Basic Attachments

An attachment declares that a concept can have associated data:

```
concept User;

struct Login {
    string nickname;
    string password;
};

attachment<User, Login> login;
```

This creates a mapping: each `User` key can have a `Login` document attached.

Recommended Pattern: Multiple Attachments

The **recommended design** uses multiple attachments per concept, each representing a separate concern:

```
// From Graph Editor: Vertex with separate concerns
concept Vertex;

// Rendering concern
struct Vertex2DAttributes {
    Position position;
};
attachment<Vertex, Vertex2DAttributes> render2DAttributes;

// Appearance concern
struct VertexVisualAttributes {
    int64 value;
    Color color;
};
attachment<Vertex, VertexVisualAttributes> visualAttributes;
```

Graph Editor Example

The Graph Editor demonstrates this pattern with **5 attachments on Graph** and **2 attachments on Vertex**:

```
namespace Graph {27c49329-a399-415c-baf0-db42949d2ba2} {

concept Graph;
concept Vertex;
concept Edge;

// Graph attachments - each is independent
attachment<Graph, GraphTopology> topology;      // Structure
attachment<Graph, GraphSelection> selection;    // UI state
attachment<Graph, GraphDescription> description; // Metadata
attachment<Graph, map<string, string>> tags;    // User-defined tags
attachment<Graph, xarray<string>> comments;     // Ordered comments

// Vertex attachments - separate concerns
attachment<Vertex, Vertex2DAttributes> render2DAttributes;
attachment<Vertex, VertexVisualAttributes> visualAttributes;

// Edge attachment
attachment<Edge, EdgeTopology> topology;

};
```

Benefits of Multiple Attachments

Benefit	Explanation
Independent evolution	Add new attachments without migrating existing data
Optional data	A Vertex can exist without visual attributes
Separation of concerns	UI state (selection) is separate from data (topology)
Partial loading	Load only the attachments you need
Clean migrations	Old code ignores new attachments

Single-Attachment Pattern

When translating an existing C++ codebase to DSM, the natural result is a **single attachment per concept** — mirroring the original C++ structures:

```
// From Raptor Editor: Single attachment per concept (translated from C++)
concept Material;

struct MaterialStandardProperties {
    string name = "MaterialStandard";
    optional<key<Thumbnail>> thumbnailKey;
    MaterialStandardType materialType = .diffuseSpecular;
    Vector diffuseColor;
    Vector ambientColor;
    Vector illuminationColor;
    float diffuseIntensity;
    float illuminationIntensity;
    optional<key<Texture>> diffuseMapKey;
    bool diffuseMapEnabled;
    // ... 50+ more fields ...
};

attachment<MaterialStandard, MaterialStandardProperties> properties;
```

Problems with Monolithic Structures

Problem	Impact
All-or-nothing	Must load entire structure even for one field
Migration pain	Adding a field requires data migration
Field bloat	Structures grow unwieldy over time
Tight coupling	Changes affect all code using the structure

Schema Evolution: Adding Features

With Multiple Attachments (Recommended)

```
// Original design
attachment<User, Login> login;

// V2: Add email separately - NO MIGRATION NEEDED
```

(continues on next page)

(continued from previous page)

```
struct ContactInfo {
    string email;
    optional<string> phone;
};
attachment<User, ContactInfo> contact;

// V3: Add avatar - NO MIGRATION NEEDED
attachment<User, blob_id> avatar;
```

Existing data is untouched. Old code continues to work.

With Single Attachment

```
// Original
struct UserProperties {
    string username;
    string password_hash;
};

// V2: Must migrate ALL existing data to add email
struct UserProperties {
    string username;
    string password_hash;
    string email;           // New field - requires migration
    optional<blob_id> avatar;  // New field - requires migration
};
```

Attachment Types

Attachments can use any DSM type as the document:

```
// Structure
attachment<Graph, GraphTopology> topology;

// Map
attachment<Graph, map<string, string>> tags;

// Collection
attachment<Graph, xarray<string>> comments;

// Simple type
attachment<User, blob_id> avatar;

// Any type (dynamic)
attachment<User, any> metadata;
```

Optional vs Required

Attachments are always **optional** at the storage level. Your code decides whether an attachment is logically required:

```
# Read an attachment
topology = commit.get(a_topology, graph_key)
```

(continues on next page)

(continued from previous page)

```
if topology.is_nil():
    # Handle missing attachment
    topology = GraphTopology()
    mutating.set(a_topology, graph_key, topology)
```

Real-World Comparison

Graph Editor (Recommended)

```
// Vertex can have:
// - render2DAttributes only (topology-only vertex)
// - visualAttributes only (invisible vertex with metadata)
// - Both (fully rendered vertex)
// - Neither (just a key reference)

attachment<Vertex, Vertex2DAttributes> render2DAttributes;
attachment<Vertex, VertexVisualAttributes> visualAttributes;
```

Raptor Editor (Single-Attachment)

```
// Surface MUST have all properties defined
// No way to have a "lightweight" surface

struct SurfaceProperties {
    string name = "Surface";
    map<key<LightingLayer>, key<Texture>> lightmaps;
    key<Mesh> meshKey;
    Vector color = {1.0, 1.0, 1.0};
    set<string> tags;
    SurfaceBillboardMode billboardMode;
    optional<key<MeshAnimation>> meshAnimationKey;
    SurfaceMirrorPlane mirrorPlane;
};

attachment<Surface, SurfaceProperties> properties;
```

Design Guidelines

Do

- **Separate concerns**: One attachment per logical domain
- **Use multiple attachments**: They're cheap and enable evolution
- **Name attachments clearly**: `topology`, `selection`, `visual` not `data`, `props`
- **Keep structures focused**: 5-10 fields per structure is ideal

Don't

- **Create monolithic structures**: Avoid 50+ field `Properties` structures
- **Mix concerns**: Don't put UI state in data structures
- **Use single attachment**: Unless you have a specific reason

Migration Path

If you inherit a single-attachment design:

1. Keep the existing `properties` attachment
2. Add new features as new attachments
3. Gradually migrate fields to new attachments
4. Deprecate unused fields (don't delete - maintains compatibility)

```
// Existing single-attachment
attachment<Vertex, VertexProperties> properties;

// New features - separate attachments
attachment<Vertex, VertexPhysics> physics;      // New feature
attachment<Vertex, VertexAnnotation> annotation; // New feature
```

What's Next

- *Function Pools* - Define operations on your data model
- *Code Generation* - Generate infrastructure from DSM

3.1.5 Function Pools

Function pools define operations on your data model. They expose business logic as callable functions accessible from C++, Python, or RPC.

Two Types of Function Pools

DSM provides two function pool types:

Type	Purpose
`function_pool`	Pure utility functions
`attachment_function_pool`	Stateful mutations

Function Pool (Pure Functions)

A `function_pool` contains stateless functions.

```
// From Graph Editor: Pure utility functions
function_pool Tools {dc9740c9-9d1d-4c1e-9caa-4c8843b91e82} {

"""Return a + b."""
int64 add(int64 a, int64 b);
```

(continues on next page)

(continued from previous page)

```
"""Return true if a is even."""
bool isEven(int64 a);

"""Return true if a > b."""
bool isGreater(any a, any b);

"""Return a random word."""
string randomWord(uint64 size);

};
```

Use cases:

- Mathematical computations
- String formatting
- Random data generation for testing
- Validation logic

Attachment Function Pool

An `attachment_function_pool` contains functions that operate via the AttachmentMutating interface

```
// From Graph Editor: Graph editing operations
attachment_function_pool ModelGraph {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e6} {

"""Create a vertex with the specified attributes."""
mutable key<Vertex> newVertex(key<Graph> graphKey, int64 value, Position position);

"""Create a new edge."""
mutable key<Edge> newEdge(key<Graph> graphKey, key<Vertex> vaKey, key<Vertex> vbKey);

"""Set the vertex color."""
mutable void setVertexColor(key<Vertex> vertexKey, Color color);

"""Set the vertex position."""
mutable void setVertexPosition(key<Vertex> vertexKey, Position position);

"""Clear the graph."""
mutable void clearGraph(key<Graph> graphKey);

};
```

The `mutable` Keyword

Functions that modify the state are marked `mutable`:

```
// Mutable: modifies state via AttachmentMutating
mutable key<Vertex> newVertex(key<Graph> graphKey, int64 value, Position position);

// Non-mutable: read-only via AttachmentGetting
set<key<Vertex>> selectedVertices(key<Graph> graphKey);
```

Graph Editor Pool Examples

Selection Pool

Manages UI selection state separately from topology:

```
// From Graph Editor: Selection management
attachment_function_pool ModelSelection {5318ad8d-79d8-498e-b080-eccf15e4a74d} {

"""Select all edges and vertices."""
mutable void selectAll(key<Graph> graphKey);

"""Deselect all edges and vertices."""
mutable void deselectAll(key<Graph> graphKey);

"""Invert the selection for edges and vertices."""
mutable void invertSelection(key<Graph> graphKey);

"""Reduce the selection to the vertex."""
mutable void setSelectionToVertex(key<Graph> graphKey, key<Vertex> vertexKey);

"""Combine the vertex-selected state with the vertex selection."""
mutable void combineSelectionWithVertex(key<Graph> graphKey, key<Vertex> vertexKey,␣
↪bool selected);

"""Delete the selected elements."""
mutable void deleteSelection(key<Graph> graphKey);

"""Return selected vertices."""
set<key<Vertex>> selectedVertices(key<Graph> graphKey);

"""Return selected edges."""
set<key<Edge>> selectedEdges(key<Graph> graphKey);

};
```

Integrity Pool

Demonstrates referential integrity repair strategies:

```
// From Graph Editor: Integrity repair operations
attachment_function_pool ModelIntegrity {95e0f859-65e0-419e-8b81-77547c73b759} {

"""Restore the graph integrity by creating missing elements."""
mutable void restoreIntegrityByCreating(key<Graph> graphKey);

"""Restore the graph integrity by removing elements."""
mutable void restoreIntegrityByDeleting(key<Graph> graphKey);

"""Restore the graph integrity by respawning elements."""
mutable void restoreIntegrityByRestoring(key<Graph> graphKey);

};
```

Raptor Editor Pool Example

Material Assignment Pool

A focused pool for a single operation:

```
// From Raptor Editor: Material assignment to surfaces
attachment_function_pool ModelSurface {fd61d936-d350-4a18-a2ca-cc7d7d3a9dc6} {

"""Assign a material to a surface."""
mutable void assign_material(
    key<AspectLayer> layerKey,
    key<Surface> surfaceKey,
    key<Material> materialKey
);

};
```

Function Documentation

Use triple-quoted strings for function documentation:

```
attachment_function_pool ModelGraph {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e6} {

"""
Create a vertex with the specified attributes.

The vertex is added to the graph topology and receives
the specified position and value.

Returns the key of the newly created vertex.
"""
mutable key<Vertex> newVertex(key<Graph> graphKey, int64 value, Position position);

};
```

Parameter Types

Functions can use any DSM type as parameters or return values:

```
attachment_function_pool Example {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e7} {

// Keys as parameters
mutable void linkItems(key<Item> a, key<Item> b);

// Structures as parameters
mutable void setTransform(key<Object> obj, Transform t);

// Collections as parameters
mutable void moveVertices(set<key<Vertex>> vertices, Position offset);

// Collections as return values
set<key<Vertex>> findConnected(key<Vertex> start);
```

(continues on next page)

(continued from previous page)

```
// Optional return values
optional<key<Vertex>> findByName(key<Graph> graph, string name);

};
```

XArray Operations

XArray uses UUID positions instead of indices for multiplayer editing:

```
// From Graph Editor: XArray comment operations
attachment_function_pool ModelGraph {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e6} {

"""Append a new comment."""
mutable void appendGraphComment(key<Graph> graphKey, string comment);

"""Insert a new comment at a specific xarray position (not an index)."""
mutable void insertGraphComment(key<Graph> graphKey, uuid position, string comment);

"""Remove a comment at a specific xarray position (not an index)."""
mutable void removeGraphComment(key<Graph> graphKey, uuid position);

"""Update the comment at a specific xarray position (not an index)."""
mutable void updateGraphComment(key<Graph> graphKey, uuid position, string comment);

};
```

Pool Namespacing

Each pool has a UUID that identifies it uniquely:

```
// Different pools for different concerns
attachment_function_pool ModelGraph {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e6} {
...
};
attachment_function_pool ModelSelection {5318ad8d-79d8-498e-b080-eccf15e4a74d} {
...
};
attachment_function_pool ModelIntegrity {95e0f859-65e0-419e-8b81-77547c73b759} {
...
};
function_pool Tools {dc9740c9-9d1d-4c1e-9caa-4c8843b91e82} {
...
};
```

Design Guidelines

Organize by Domain

Group related functions into pools by concern:

- `ModelGraph`: Operations on graph topology
- `ModelSelection`: UI selection state management

- `ModelIntegrity`: Validation and repair operations
- `Tools`: Pure utility functions

Keep Functions Focused

Each function should do one thing:

```
// Good: Focused functions
mutable void setVertexColor(key<Vertex> vertexKey, Color color);
mutable void setVertexPosition(key<Vertex> vertexKey, Position position);

// Avoid: Multi-purpose functions
mutable void updateVertex(key<Vertex> vertexKey, optional<Color> color,
optional<Position> position, optional<int64> value);
```

Use Queries for Read-Only Operations

Non-mutable functions can be called without a commit context:

```
// Query: doesn't need commit context
set<key<Vertex>> selectedVertices(key<Graph> graphKey);

// Mutation: requires commit context
mutable void selectAll(key<Graph> graphKey);
```

What's Next

- *Code Generation* - Generate infrastructure from DSM
- *Architecture* - Application structure with generated code

3.1.6 Code Generation

Kibo generates application infrastructure from DSM definitions. The developer focuses on business logic while Kibo handles repetitive plumbing.

Why Generate Code?

The Amplification Effect

For every line of DSM, Kibo generates approximately 57 lines of C++ and Python:

Project	DSM Lines	Generated Lines	Ratio
Graph Editor (Pilot)	273	15,592	57x
Raptor Editor (Industrial)	2,003	126,182	63x

What Gets Generated vs What You Write

Category	Who Writes It	Examples
DSM definitions	Developer	`Graph.dsm`, `Pool_ModelGraph.dsm`
Infrastructure	Kibo	Serialization, persistence, accessors
Business logic	Developer	Domain rules, algorithms, UI

The generated infrastructure handles:

- Type-safe data structures
- Attachment accessors (get/set/keys via abstract interfaces)
- Database persistence layer
- Binary and JSON serialization
- Python bindings
- RPC marshalling

The Double Reality

Kibo creates two coexisting realities that are bridged transparently:

Developer Reality

What you see and code:

- Static C++ types with IDE support
- STL containers (`std::vector`, `std::map`)
- Compile-time type errors
- Idiomatic, readable code

```
// Developer writes natural C++
MyStruct data{"hello", {1, 2, 3}};
data.items.push_back(42);
```

Runtime Reality

What happens behind the scenes:

- Metadata-driven operations
- Reference semantics with `shared_ptr`
- Automatic serialization
- Python interoperability
- RPC without manual marshalling

```
┌──────────────────────────────────┐
│  Developer Reality               │
│  Static C++ types, Python types  │
│  MyStruct data{"hello", ...};    │
```

(continues on next page)

(continued from previous page)

```
└──────────────────────────────────────┘
═══════════════════════════════════════

┌──────────────────────────────────────┐
│  Runtime Reality                     │
│  Dynamic metadata, shared_ptr        │
│  Serialization, RPC                  │
└──────────────────────────────────────┘
```

The developer writes idiomatic C++ and never sees the runtime reality—Kibo generates the bridge code.

Generated Layers

1. Data Layer

Generates type-safe business types:

- Concept keys (`VertexKey`, `EdgeKey`)
- Structures as C++ classes
- Enumerations
- Hash functions for containers

```
// DSM input
concept Vertex;
struct Position { float x; float y; };
```

```
// Generated output
class VertexKey { ... };
struct Position { float x; float y; };
```

2. Attachments Layer

Generates stateful accessors for attachments:

```
// Generated accessor API
auto topology = Attachments::Graph_Topology::get(attachmentGetting, graphKey);
Attachments::Graph_Topology::set(attachmentMutating, graphKey, newTopology);
Attachments::Graph_Topology::unionVertexKeys(attachmentMutating, graphKey, {key});
```

3. Database Layer

Generates persistence without writing SQL:

- SQLite backend for local storage
- Remote database client for RPC
- CRUD operations for all attachments

4. Serialization Layer

Generates two-way converters:

- Binary serialization (efficient, compact)
- JSON serialization (human-readable)
- C++ ↔ Viper Value bridge

5. Function Pool Layer

Generates function exposure:

- Marshalling between C++ and Viper
- Marshalling Python and Viper
- RPC server and client stubs

6. Python Package Layer

Generates a complete Python package:

```
# Generated Python package
import ge.data as ged
import ge.attachments as gea

# Use generated types
vertex_key = ged.Graph_VertexKey.create()
position = ged.Position()
position.x = 10.0
position.y = 20.0

# Use generated accessors
gea.graph_vertex_render2d_attributes_set(
    attachment_mutating, vertex_key, position
)
```

The Bridge Pattern

The **Bridge** connects your business logic to DSM-declared functions:

```
┌──────────────────────────────────────────────┐
│  DSM (Pool_ModelGraph.dsm)                   │
│  mutable key<Vertex> newVertex(...)          │
└──────────────────────┬───────────────────────┘
                       │ generates
┌──────────────────────▼───────────────────────┐
│  Bridge.hpp (GENERATED - contract)           │
│  VertexKey new_vertex(...);                  │
└──────────────────────┬───────────────────────┘
                       │ implements
┌──────────────────────▼───────────────────────┐
│  Bridge.cpp (BUSINESS - hand-written)        │
│  return Model::Vertex::add(...);             │
```

(continues on next page)

(continued from previous page)

```
└──────────────────────────────────────────┘
                      │
                      │ exposes via
                      ▼
┌──────────────────────────────────────────┐
│  FunctionPools.cpp (GENERATED)           │
│  Value ↔ C++ marshalling                 │
└──────────────────────────────────────────┘
                      │
          ┌───────────┴──────────┐
          ▼                      ▼
   Python Local            Remote RPC
```

Example Workflow

1. DSM Declaration:

```
attachment_function_pool ModelGraph {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e6} {
    mutable key<Vertex> newVertex(key<Graph> graphKey, int64 value, Position␣
↪position);
};
```

2. Bridge Header (Generated):

```
// GE_AttachmentFunctionPoolBridges.hpp - GENERATED by Kibo
namespace GE::AttachmentFunctionPoolBridges {
namespace ModelGraph {

Graph::VertexKey new_vertex(std::shared_ptr<Viper::AttachmentMutating> const &␣
↪attachmentMutating,
                            Graph::GraphKey const & graphKey,
                            std::int64_t value,
                            Graph::Position const & position);

}
}
```

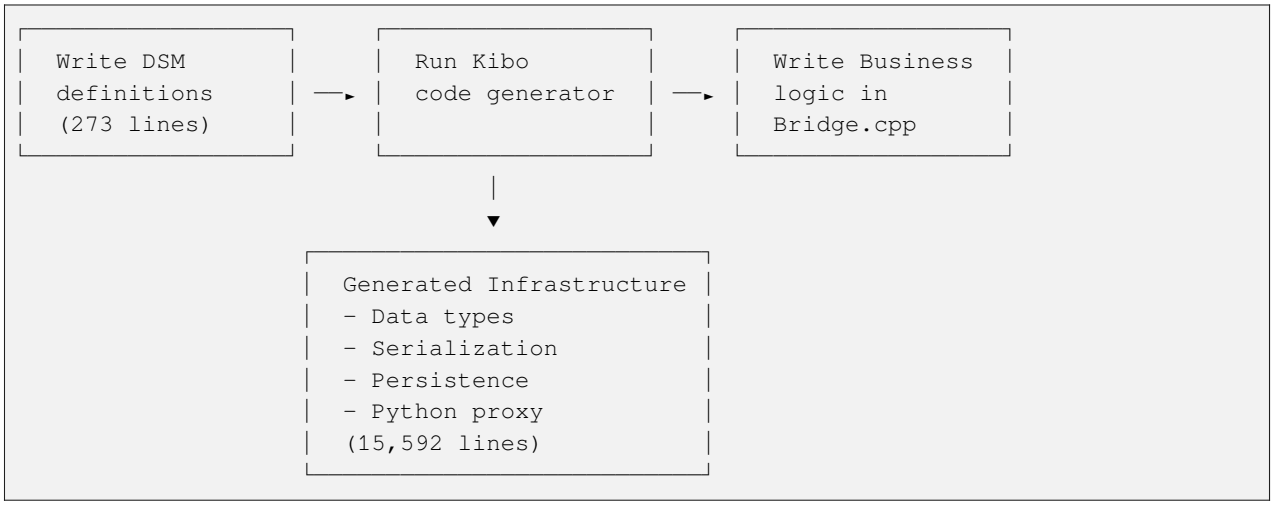
3. Bridge Implementation (Business):

```
// GE_AttachmentFunctionPoolBridges.cpp - HAND-WRITTEN
Graph::VertexKey new_vertex(std::shared_ptr<Viper::AttachmentMutating> const &␣
↪attachmentMutating,
                            Graph::GraphKey const & graphKey,
                            std::int64_t value,
                            Graph::Position const & position) {
    return Model::Vertex::add(attachmentMutating, graphKey, value, position,
                              Model::Random::makeColor());
}
```

4. Call from Python:

```
# Automatically available in Python
vertex_key = pool.new_vertex(commit, graph_key, 42, position)
```

Workflow Summary



Benefits

Benefit	Explanation
Less code to write	57x amplification means 57x less code to maintain
Type safety	Generated code is type-safe, reducing runtime errors
Consistency	All serialization and persistence follows the same patterns
Multi-language	Same DSM generates C++ and Python code
Evolution	Change DSM, regenerate, and infrastructure updates automatically

What's Next

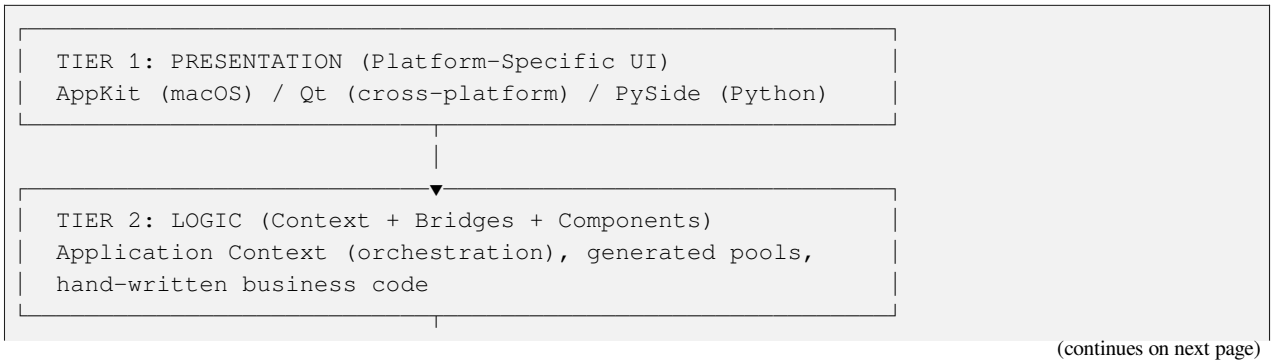
- *Architecture* - Application structure with generated code
- *Toolchain: Kibo* - Using the Kibo code generator

3.1.7 Application Architecture

This chapter describes how to structure applications built with DSM and Viper. The architecture separates concerns into layers, enabling code reuse across platforms and languages.

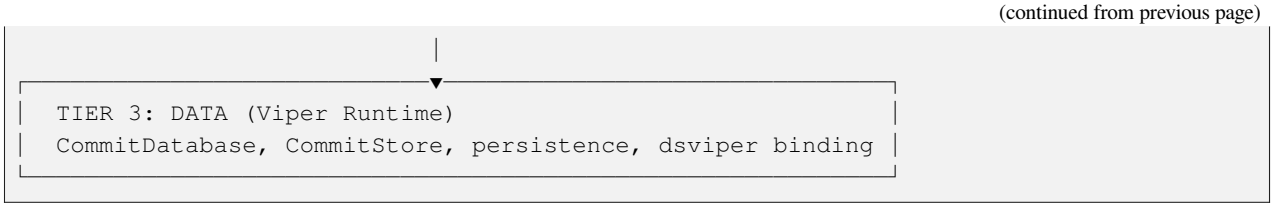
The 3-Tier Architecture

Applications built with Viper follow a 3-tier pattern:



(continues on next page)

(continued from previous page)

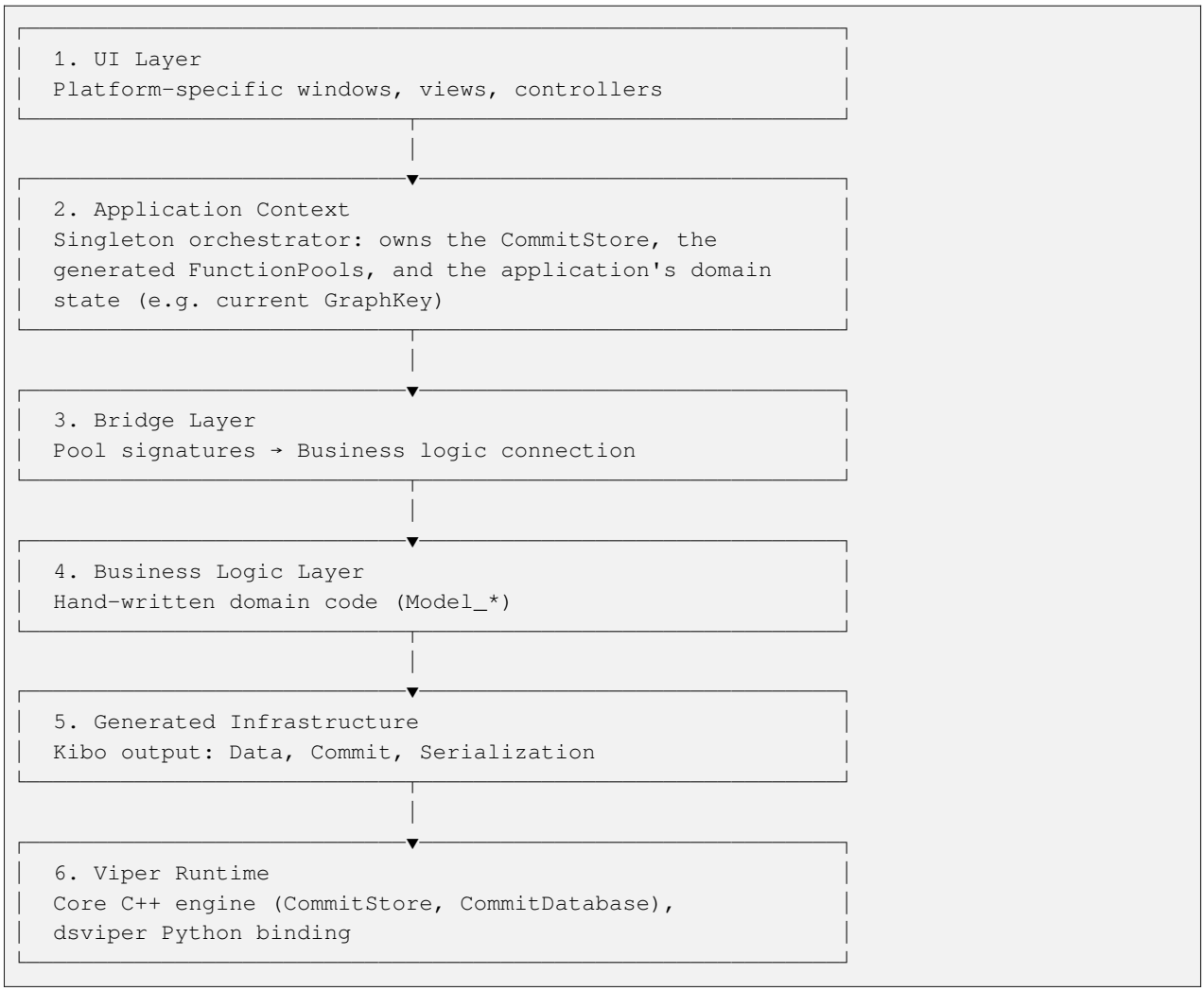


Tier Responsibilities

Tier	Responsibility	Platform-Specific?
Presentation	UI framework, windows, dialogs	Yes
Logic	Application Context, bridges, business components	Partially
Data	Viper runtime (CommitDatabase, CommitStore), persistence	No (shared)

The Full Application Stack

For business applications with domain logic, the stack expands:



Layer Details

Layer 1: UI

Platform-specific presentation code:

- Window management
- Menu and toolbar actions
- Dialog boxes
- Framework-native widgets

This layer changes for each target platform but follows the same logical structure.

Layer 2: Application Context

The **Context** is the application-level singleton that orchestrates the runtime. It is *not* a subclass of `Viper::CommitStore`: it **owns** a `CommitStore` (provided by the Viper runtime) and adds the application-specific surface around it — the generated `FunctionPool` / `AttachmentFunctionPool` instances, the domain state (e.g. the current graph), and the action dispatch.

Responsibilities of the Context:

- Own the `CommitStore` (which itself holds the `CommitDatabase` and the current `CommitState`)
- Hold the generated tool / model / attachment `FunctionPools`
- Hold application-level domain state (e.g. `graphKey`)
- Open / close / load the database
- Dispatch user actions through `CommitMutableState`
- Expose framework-agnostic notifications via `CommitStoreNotifying`

```
// Context pattern (C++) — see com.digitalsubstrate.ge / GE_Context.hpp
namespace GE {

class Context final {
public:
    static std::shared_ptr<Context> Instance();

    // Database lifecycle (owned via the store)
    void use(std::shared_ptr<Viper::CommitDatabase> const & database);
    void close();
    void load();
    static std::shared_ptr<Viper::CommitDatabase> createDatabase(
        std::filesystem::path const & filePath);

    // Domain operations
    void newGraph();
    void useNewGraph(std::string const & label);
    void dispatchAction(std::string const & label,
                        std::shared_ptr<ContextAction> const & action);

    // Owned components
    std::shared_ptr<Viper::CommitStore> store;
```

(continues on next page)

(continued from previous page)

```
    std::shared_ptr<Viper::FunctionPool>           tools;
    std::shared_ptr<Viper::AttachmentFunctionPool> modelGraph;
    std::shared_ptr<Viper::AttachmentFunctionPool> modelSelection;
    std::shared_ptr<Viper::AttachmentFunctionPool> modelIntegrity;
    std::shared_ptr<Viper::AttachmentFunctionPool> attachments;

    // Domain state
    GE::Graph::GraphKey graphKey;
};

} // namespace GE
```

Action dispatch always goes through the store's `CommitMutableState`, so undo/redo and notifications stay coherent:

```
void Context::dispatchAction(std::string const & label,
                             std::shared_ptr<ContextAction> const & action) {
    auto const ms{store->mutableState()};
    action->run(ms);
    store->commitMutations(label, ms);
}
```

The previous "Store-as-application-singleton" pattern is superseded: the generic store concerns (database, state, undo/redo, notifications) live in `Viper::CommitStore`, and the application-specific concerns (which pools, which domain state, which actions) live in the application's `Context`.

Layer 3: Bridges

Bridges connect generated pool signatures to your business logic:

```
// GE_AttachmentFunctionPoolBridges.hpp (GENERATED)
namespace GE::AttachmentFunctionPoolBridges {
namespace ModelGraph {

Graph::VertexKey new_vertex(std::shared_ptr<Viper::AttachmentMutating> const &␣
↪attachmentMutating,
                            Graph::GraphKey const & graphKey,
                            std::int64_t value,
                            Graph::Position const & position);

}
}

// GE_AttachmentFunctionPoolBridges.cpp (HAND-WRITTEN)
Graph::VertexKey new_vertex(std::shared_ptr<Viper::AttachmentMutating> const &␣
↪attachmentMutating,
                            Graph::GraphKey const & graphKey,
                            std::int64_t value,
                            Graph::Position const & position) {
    return Model::Vertex::add(attachmentMutating, graphKey, value, position,
                              Model::Random::makeColor());
}
```

Layer 4: Business Logic

Your domain-specific code:

```
// GE_Model_Vertex.cpp
VertexKey add(std::shared_ptr<AttachmentMutating> const & attachmentMutating,
              GraphKey const & graphKey,
              std::int64_t const & value,
              Position const & position,
              Color const & color) {

    auto const vertexKey{create(attachmentMutating, value, position, color)};
    Attachments::Graph_Topology::unionVertexKeys(attachmentMutating, graphKey,
↪{vertexKey});

    return vertexKey;
}
```

Layer 5: Generated Infrastructure

Kibo generates this layer from your DSM:

- Data types (`*_Data.cpp`)
- Commit accessors (`*_Commit.cpp`)
- Serialization (`*_Reader.cpp`, `*_Writer.cpp`)
- Database persistence (`*_Database.cpp`)
- Python proxy

Layer 6: Viper Runtime

The core C++ engine providing:

- Type and value system
- CommitDatabase persistence
- Stateful commits
- Python binding (dsviper)

The Notifier Pattern

UI frameworks have different notification mechanisms. The **Notifier** pattern bridges this gap:

```
CommitStore (C++ core)
        |
        ▼
CommitStoreNotifying (interface)
        |
        ▼
DSCommitStoreNotifier (framework adapter)
        |
        ▼
Framework-specific notifications
```

Common Notifications

All implementations expose the same signals:

Signal	Purpose
`database_did_open`	Database opened successfully
`database_did_close`	Database closed
`state_did_change`	State changed (undo/redo available)
`definitions_did_change`	Schema updated
`dispatch_error`	Error occurred

Multi-Language Support

The architecture supports multiple languages:

Layer	C++ Application	Python Application
UI	AppKit/Qt C++	PySide/Qt Python
Context	C++ singleton (owns CommitStore)	Python class (owns store)
Business	C++	Python or dsviper
Infrastructure	C++ generated	Python generated
Runtime	Viper C++ (CommitStore, …)	dsviper

Python Integration

The `dsviper` module exposes the entire Viper runtime, including `CommitStore` and `CommitDatabase`. A Python application's `Context` follows the same pattern as the C++ one — it instantiates a `CommitStore` and wires the generated pools around it:

```
import dsviper

database = dsviper.CommitDatabase.create("path/to/data.cdb",
                                         "An Application Database")
store = dsviper.CommitStore.make()
store.set_database(database)

# Undo/Redo operations
if store.can_undo():
    store.undo()
```

Design Principles

Separation of Concerns

Each layer has a single responsibility:

- **UI**: Only presentation, no business logic
- **Context**: Orchestration only — owns the `CommitStore` and the pools, does not reimplement them
- **Bridges**: Connection, not logic
- **Business**: Domain rules, not infrastructure
- **Generated**: Infrastructure, not business

Platform Isolation

Platform-specific code lives only in Layer 1. The other layers are shared or generated.

Generated vs Hand-Written

Generated (Kibo)	Hand-Written
Data types	Business logic
Serialization	Bridges
Persistence	Store orchestration
Pool marshalling	UI layer

Minimal Porting Effort

To port to a new platform:

1. Implement the Notifier adapter (~50 lines)
2. Create UI components matching existing patterns
3. Wire up the main window

The business logic and infrastructure remain unchanged.

Summary

Principle	Implementation
Separation	6 distinct layers with clear responsibilities
Reuse	Layers 3-6 shared across platforms
Generation	Kibo generates infrastructure (Layer 5)
Multi-language	Same architecture for C++ and Python
Platform isolation	Only Layer 1 is platform-specific

This architecture enables building complex applications that work across platforms while minimizing duplicated code.

What's Next

- *Python Guide* - Use the dsviper Python API
- *Toolchain* - Master the development tools

3.1.8 DSM Samples

This section presents two real-world applications that demonstrate DSM modeling patterns at different scales and design philosophies.

Application	Pattern	Description
Graph Editor	**Recommended** (multi-attachment)	Graph topology editor with 5 DSM files. Demonstrates best practices with multiple attachments per concept, separating concerns cleanly.
Raptor Editor	Single-attachment (translated from C++)	Professional 3D visualization with 20 DSM files. Production-scale example showing materials, lighting, cameras, and animation.

Pattern Comparison

Recommended Pattern (Graph Editor)

Each concept uses multiple attachments to separate concerns:

```
// Topology is separate from selection, which is separate from rendering
attachment<Graph, GraphTopology> topology;
attachment<Graph, GraphSelection> selection;
attachment<Vertex, Vertex2DAttributes> render2DAttributes;
attachment<Vertex, VertexVisualAttributes> visualAttributes;
```

Benefits:

- Adding a new concern (e.g., `physicsAttributes`) requires zero migration
- Each attachment is optional and independent
- A Vertex can exist without render attributes (topology-only)

Single-Attachment Pattern (Raptor Editor)

Each concept has a single monolithic `properties` attachment:

```
attachment<Camera, CameraProperties> properties;
attachment<Material, MaterialProperties> properties;
```

Trade-offs:

- Simpler initial design
- Harder to evolve: adding optional features requires struct changes
- All properties bundled together even when some are unused

Note

New projects should follow the **recommended pattern**. The single-attachment pattern is documented here because it demonstrates how DSM can adapt to an existing C++ codebase.

Graph Editor

Files: 5 | Namespace: Graph | Pattern: Recommended (multiple attachments, function pools)

Graph topology editor demonstrating best practices.

Graph.dsm

Domain: Graph, Vertex, Edge

This file defines the core data model for a graph editor application. It demonstrates the **recommended Viper pattern** with multiple attachments per concept, separating concerns cleanly.

Data Model Design

- **Three concepts** (Graph, Vertex, Edge) represent the topological entities
- **Attachments separate concerns**: topology (structure), selection (UI state), description (metadata), render2DAttributes (visualization), visualAttributes (appearance)
- **Key insight**: A Vertex can exist without render attributes (topology-only) or without visual attributes (invisible vertex). Each attachment is optional and independent.

Why Multiple Attachments Matter

- `topology` attachment stores which vertices/edges exist in the graph
- `selection` attachment tracks UI selection state separately from topology
- `visualAttributes` and `render2DAttributes` handle rendering independently
- Adding a new concern (e.g., `physicsAttributes`) requires zero migration - just add a new attachment

This is the **anti-monolithic pattern**: instead of one giant `VertexProperties` struct, each concern lives in its own attachment.

```
// Types definitions
namespace Graph {27c49329-a399-415c-baf0-db42949d2ba2} {

"""A graph."""
concept Graph;

"""A vertex in a graph."""
concept Vertex;

"""An edge in a graph."""
concept Edge;

"""The selected vertices and edges."""
struct GraphSelection {
    set<key<Vertex>> vertexKeys;
    set<key<Edge>> edgeKeys;
};

"""An edge in the graph topology."""
struct EdgeTopology {
    key<Vertex> vaKey;
    key<Vertex> vbKey;
};

"""The vertices and edges of the graph topology."""
struct GraphTopology {
    set<key<Vertex>> vertexKeys;
```

(continues on next page)

(continued from previous page)

```
    set<key<Edge>> edgeKeys;
};

"""A Rectangle."""
struct Rectangle {
    """the x origin."""
    float x;
    """the y origin."""
    float y;
    """the width."""
    float w;
    """the height."""
    float h;
};

"""The descriptive information."""
struct GraphDescription {
    string name;
    string author;
    string createDate;
};

"""A Position."""
struct Position {
    float x;
    float y;
};

"""The attributes used to render a topological vertex in 2D."""
struct Vertex2DAttributes {
    Position position;
};

"""An RGB Color."""
struct Color {
    float red;
    float green;
    float blue;
};

"""The visual attributes of a vertex."""
struct VertexVisualAttributes {
    int64 value;
    Color color;
};

};

// Attachments definitions
namespace Graph {27c49329-a399-415c-baf0-db42949d2ba2} {
"""The topology of a graph."""
attachment<Graph, GraphTopology> topology;
```

(continues on next page)

(continued from previous page)

```
"""A collection of tags for a graph."""
attachment<Graph, map<string, string>> tags;

"""A collection of comments for a Graph."""
attachment<Graph, xarray<string>> comments;

"""The selected vertices of a graph."""
attachment<Graph, GraphSelection> selection;

"""The descriptive information of a graph."""
attachment<Graph, GraphDescription> description;

"""The render attributes used to render a vertex of a graph in 2D."""
attachment<Vertex, Vertex2DAttributes> render2DAttributes;

"""The visual attributes used to render a vertex of a graph."""
attachment<Vertex, VertexVisualAttributes> visualAttributes;

"""An Edge in the graph topology."""
attachment<Edge, EdgeTopology> topology;

};
```

Pool_ModelGraph.dsm

Domain: Pool: ModelGraph

This is an **attachment_function_pool** - functions that modify the AttachmentMutableState.

Function Categories

- **Topology operations**: `newVertex`, `newEdge`, `clearGraph` - basic graph manipulation
- **Property setters**: `setVertexColor`, `setVertexPosition`, `setVertexValue` - modify vertex attachments
- **Batch operations**: `moveVertices`, `randomGraph` - operate on multiple elements
- **Metadata**: `setGraphLabel`, `setGraphTag`, `appendGraphComment` - graph-level annotations
- **XArray operations**: `insertGraphComment`, `removeGraphComment`, `updateGraphComment` - demonstrate ordered collection with UUID-based positions (not indices)

Testing/Demo Functions

These functions are intentionally broken for demonstrations:

- `graphWithError` - throws during commit to test error handling
- `graphWithMissingVertex` - creates dangling edge references
- `deleteSelectionBugged` - deletes vertices without cleaning edges
- `killer` - stress test that creates many mutations

Note

All `mutable` functions operate within a mutation context (AttachmentMutating). Non-mutable functions (queries) only need a reading context (AttachmentGetting).

```
"""This pool provides access to functions used to edit the graph topology."""
attachment_function_pool ModelGraph {9bdcbb5b-76e9-426f-b8a6-a10ed2d949e6} {

"""Append a new comment."""
mutable void appendGraphComment(key<Graph> graphKey, string comment);

"""Clear the graph."""
mutable void clearGraph(key<Graph> graphKey);

"""Delete a vertex without deleting connected edges."""
mutable void deleteSelectionBugged(key<Graph> graphKey);

"""Throw an error while creating a graph."""
mutable void graphWithError(key<Graph> graphKey);

"""Create a graph with missing vertices to illustrate graph integrity."""
mutable void graphWithMissingVertex(key<Graph> graphKey);

"""Create a graph with missing vertex property to illustrate graph integrity."""
mutable void graphWithMissingVertexProperties(key<Graph> graphKey);

"""Insert a new comment at a specific xarray position (not an index)."""
mutable void insertGraphComment(key<Graph> graphKey, uuid position, string comment);

"""Create mutations that destroy the usability of the Commit."""
mutable void killer(key<Graph> graphKey, uint64 vertexCount);

"""Move the vertices by the offset."""
mutable void moveVertices(set<key<Vertex>> vertexKeys, Position offset);

"""Create a new edge."""
mutable key<Edge> newEdge(key<Graph> graphKey, key<Vertex> vaKey, key<Vertex> vbKey);

"""Create a vertex with the specified attributes."""
mutable key<Vertex> newVertex(key<Graph> graphKey, int64 value, Position position);

"""Append a random comment."""
mutable void randomComment(key<Graph> graphKey);

"""Create a random edge or raise if the graph is complete."""
mutable key<Edge> randomEdge(key<Graph> graphKey);

"""Create a new random graph in rect."""
mutable void randomGraph(key<Graph> graphKey, uint64 vertexCount, uint64 edgeCount,␣
↪Rectangle rect);

"""Insert a new random tag."""
```

(continues on next page)

(continued from previous page)

```
mutable void randomTag(key<Graph> graphKey);

"""Create a random vertex."""
mutable key<Vertex> randomVertex(key<Graph> graphKey, Rectangle rect);

"""Remove a comment at a specific xarray position (not an index)."""
mutable void removeGraphComment(key<Graph> graphKey, uuid position);

"""Set the label of the graph description."""
mutable void setGraphLabel(key<Graph> graphKey, string label);

"""Replace or insert the tag."""
mutable void setGraphTag(key<Graph> graphKey, string key, string value);

"""Set the vertex color."""
mutable void setVertexColor(key<Vertex> vertexKey, Color color);

"""Set the vertex position."""
mutable void setVertexPosition(key<Vertex> vertexKey, Position position);

"""Set the vertex value."""
mutable void setVertexValue(key<Vertex> vertexKey, int64 value);

"""Remove the tag."""
mutable void unsetGraphTags(key<Graph> graphKey, set<string> keys);

"""Update the comment at a specific xarray position (not an index)."""
mutable void updateGraphComment(key<Graph> graphKey, uuid position, string comment);

"""Update a tag."""
mutable void updateGraphTag(key<Graph> graphKey, string key, string value);

"""Test no args."""
mutable void noArgs();

};
```

Pool_ModelIntegrity.dsm

Domain: Pool: ModelIntegrity

This pool demonstrates **referential integrity repair** - what happens when the graph has dangling references (edges pointing to deleted vertices).

Three Repair Strategies

- `restoreIntegrityByCreating` - recreate missing vertices (preserve topology)
- `restoreIntegrityByDeleting` - remove edges with dangling references (preserve consistency)
- `restoreIntegrityByRestoring` - respawn deleted elements from history

Why This Matters

In Viper, concepts (Vertex, Edge) are identified by keys. If you delete a Vertex without updating Edge references, edges become "orphaned". This pool shows how to detect and repair such inconsistencies.

Use case: After running `deleteSelectionBugged` from ModelGraph pool (which intentionally creates orphans), use these functions to restore graph integrity.

```
"""This pool provides access functions used to restore the integrity of the graph."""
attachment_function_pool ModelIntegrity {95e0f859-65e0-419e-8b81-77547c73b759} {

"""Restore the graph integrity by creating missing elements."""
mutable void restoreIntegrityByCreating(key<Graph> graphKey);

"""Restore the graph integrity by removing elements."""
mutable void restoreIntegrityByDeleting(key<Graph> graphKey);

"""Restore the graph integrity by respawning elements."""
mutable void restoreIntegrityByRestoring(key<Graph> graphKey);

"""Restore attributes of the selected vertices."""
mutable void restoreVertexSelection(key<Graph> graphKey);

};
```

Pool_ModelSelection.dsm

Domain: Pool: ModelSelection

This pool manages **UI selection state** - which vertices and edges are currently selected by the user. Selection is stored in the `Graph.selection` attachment, separate from topology.

Selection Operations

- **Set operations**: `selectAll`, `deselectAll`, `invertSelection` - bulk selection changes
- **Element-specific**: `setSelectionToVertex`, `setSelectionToEdge` - single element selection
- **Combine mode**: `combineSelectionWithVertex/Edge` - add/remove from current selection (shift-click behavior)
- **Queries**: `selectedVertices`, `selectedEdges` - return current selection (non-mutable)

> **Note**
>
> `deleteSelection` operates on selection then modifies topology - demonstrating how UI state drives data mutations.

```
"""This pool provides access to functions used to edit the selection."""
attachment_function_pool ModelSelection {5318ad8d-79d8-498e-b080-eccf15e4a74d} {

"""Combine the edge-selected state with the edge selection."""
mutable void combineSelectionWithEdge(key<Graph> graphKey, key<Edge> edgeKey, bool␣
↪selected);
```

(continues on next page)

(continued from previous page)

```
"""Combine the vertex-selected state with the vertex selection."""
mutable void combineSelectionWithVertex(key<Graph> graphKey, key<Vertex> vertexKey,␣
↪bool selected);

"""Delete the selected elements."""
mutable void deleteSelection(key<Graph> graphKey);

"""Deselect all edges and vertices."""
mutable void deselectAll(key<Graph> graphKey);

"""Deselect all edges."""
mutable void deselectAllEdges(key<Graph> graphKey);

"""Deselect all vertices."""
mutable void deselectAllVertices(key<Graph> graphKey);

"""Increment the value of selected vertices."""
mutable void incrementVertexValue(key<Graph> graphKey, int64 value);

"""Invert the edge selection."""
mutable void invertEdgesSelection(key<Graph> graphKey);

"""Invert the selection for edges and vertices."""
mutable void invertSelection(key<Graph> graphKey);

"""Invert the vertex selection."""
mutable void invertVerticesSelection(key<Graph> graphKey);

"""Select all edges and vertices."""
mutable void selectAll(key<Graph> graphKey);

"""Select all edges."""
mutable void selectAllEdges(key<Graph> graphKey);

"""Select all vertices."""
mutable void selectAllVertices(key<Graph> graphKey);

"""Return selected edges."""
set<key<Edge>> selectedEdges(key<Graph> graphKey);

"""Return selected vertices."""
set<key<Vertex>> selectedVertices(key<Graph> graphKey);

"""Reduce the selection to the specified edges and vertices."""
mutable void setSelection(key<Graph> graphKey, set<key<Vertex>> vertexKeys, set<key
↪<Edge>> edgeKeys);

"""Reduce the edge selection to the edge."""
mutable void setSelectionToEdge(key<Graph> graphKey, key<Edge> edgeKey);

"""Reduce the selection to the vertex."""
```

(continues on next page)

(continued from previous page)

```
mutable void setSelectionToVertex(key<Graph> graphKey, key<Vertex> vertexKey);

};
```

Pool_Tools.dsm

Domain: Pool: Tools

This is a **function_pool** (not attachment_function_pool) - pure utility

Difference from attachment_function_pool

- `function_pool`: Pure functions
- `attachment_function_pool`: Functions that modify the AttachmentMutableState

Utility Functions

- `add`, `isEven`, `isGreater` - basic arithmetic/comparison (demonstration purposes)
- `randomWord` - generate random strings for testing

Use case: These functions are stateless and available in RPC context for client-side computations or test data generation (no AttachmentMutating context required).

```
"""This pool provides access to the various utility functions."""
function_pool Tools {dc9740c9-9d1d-4c1e-9caa-4c8843b91e82} {

"""Return a + b."""
int64 add(int64 a, int64 b);

"""Return true if a is even."""
bool isEven(int64 a);

"""Return true if a > b."""
bool isGreater(any a, any b);

"""Return a random word."""
string randomWord(uint64 size);

};
```

Raptor Editor

Files: 20 | Namespace: Raptor | Pattern: Single-attachment (translated from C++ codebase)

3D visualization data model for Raptor Editor.

Overview

Raptor Editor is a **stress test application** for Commit Engine performance and blob management. This DSM defines the complete data model for:

- **Scene composition**: Models, Surfaces, Geometry Layers, Position Layers
- **Materials**: Multi-layer shading (diffuse, specular, illumination), PBR materials, car paint (AxF)
- **Lighting**: Spot lights, area lights, directional lights, HDR environments
- **Camera**: Perspective/orthographic, depth of field, motion blur
- **Animation**: Timelines, keyframes, Bezier camera paths, kinematics

Note

Single-attachment pattern: This model uses **one attachment per concept** (named `properties`). It was translated directly from an existing C++ codebase, where data structures already had this monolithic form. This demonstrates that DSM can adapt to an existing codebase. For new projects, the recommended pattern (see *Graph Editor*) uses **multiple attachments per concept** to separate concerns.

Pool_Surfaces.dsm

Domain: Pool: ModelSurface

Minimal function pool for **material assignment**. In Raptor, surfaces are 3D geometry that can have different materials per AspectLayer (e.g., "showroom" vs "outdoor" lighting scenarios).

The single function `assign_material` links a Surface to a Material within a specific AspectLayer context - enabling the same model to render with different material configurations.

```
"""This pool is dedicated to Surface"""
attachment_function_pool ModelSurface {fd61d936-d350-4a18-a2ca-cc7d7d3a9dc6} {

"""assign a material to a surface"""
mutable void assign_material(key<AspectLayer> layerKey, key<Surface> surfaceKey, key
↪<Material> materialKey);

};
```

Pool_Tools.dsm

Domain: Pool: Tools

Pure utility **function_pool**. Provides helper functions for testing and UI:

- `randomColor`, `randomWord` - test data generation
- `userName` - current user for metadata
- `add`, `isEven` - basic arithmetic demos

```
"""This pool provides access to the various utility functions."""
function_pool Tools {ac3b7779-e0bf-46f8-95a1-bcf1df164022} {

"""Return a + b."""
int64 add(int64 a, int64 b);
```

(continues on next page)

(continued from previous page)

```
"""Return true if a is even."""
bool isEven(int64 a);

"""Return a random color."""
Vector randomColor();

"""Return a random word."""
string randomWord();

"""Return the current user name."""
string userName();

};
```

Raptor_BezierPath.dsm

Domain: BezierPath

Defines **smooth camera paths** for cinematic animations. A BezierPath is a spline curve that cameras can follow during timeline playback.

Structure

- `BezierPathProperties`: name, color (for viewport display), vertices list
- `BezierPathVertex`: control point with `leftTangent` and `rightTangent` for smooth interpolation
- `isClosed`: loop the path or stop at end
- `invertEvaluationDirection`: reverse camera travel direction

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept BezierPath;

struct BezierPathProperties {
    string name = "BezierPath";
    Vector color = {1.0, 1.0, 1.0};
    float startAbscissa;
    bool invertEvaluationDirection;
    bool isClosed;
    vector<BezierPathVertex> vertices;
};

struct BezierPathVertex {
    Vector point;
    Vector leftTangent;
    Vector rightTangent;
    bool areTangentLinkedToPoint;
    bool areTangentLinked;
```

(continues on next page)

(continued from previous page)

```
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<BezierPath, BezierPathProperties> properties;

};
```

Raptor_Camera.dsm

Domain: Camera

Defines the **virtual camera** for rendering. Cameras capture the scene from a specific viewpoint with realistic optical properties.

Key Properties

- `PointOfView`: position, target, up vector (defines view matrix)
- `CameraOpticalProperties`: focal length, sensor size, depth of field, aperture
- `CameraProjection`: perspective vs orthographic
- `lensShiftProperties`: tilt-shift photography simulation (architecture rendering)

Note

Camera has TWO attachments (`properties` + `lensShiftProperties`) - a deviation from the strict single-attachment pattern, showing that even translated models sometimes need multiple attachments for optional features.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Camera;

struct CameraProperties {
    string name = "Camera";
    PointOfView pointOfView;
    CameraOpticalProperties opticalProperties;
    optional<key<Sensor>> sensorKey;
    bool exposedInConfiguration = true;
};

struct CameraDepthOfField {
    bool enabled;
    float aperture;
    int32 sampleCount = 32;
```

(continues on next page)

(continued from previous page)

```
};

struct CameraDepthRange {
    CameraDepthRangePolicy policy = .lookAtPointBasedInterest;
    float zNear = 0.1;
    float zFar = 100.0;
};

enum CameraDepthRangePolicy {
    frustumBased,
    fixedDepthRange,
    lookAtPointBasedInterest,
    useGlobalPolicy
};

enum CameraFovType {
    x,
    y
};

struct CameraMotionBlur {
    bool enabled;
    bool objectMotionBlur;
    float simulatedFps = 24.0;
};

struct CameraOpticalProperties {
    CameraFovType fovType = .y;
    float fov = 0.785398;
    CameraOrientation orientation;
    CameraDepthOfField depthOfField;
    CameraMotionBlur motionBlur;
    CameraDepthRange depthRange;
};

enum CameraOrientation {
    landscape,
    portrait
};

struct LensShiftProperties {
    float lensShiftX;
    float lensShiftY;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Camera, CameraProperties> properties;
attachment<Camera, LensShiftProperties> lensShiftProperties;
```

(continues on next page)

(continued from previous page)

```
};
```

Raptor_ClippingPlane.dsm

Domain: ClippingPlaneGroup

Defines clipping planes for sectional views of 3D models.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept ClippingPlaneGroup;
struct ClippingPlaneGroupProperties {
    vector<ClippingPlaneGroupPlane> planes;
    bool isClippingEnabled;
    ClippingPlaneGroupBackfaceCullPolicy backfaceCullPolicy;
    bool isSurfaceTagsEnabled;
    set<string> surfaceTags;
};

enum ClippingPlaneGroupBackfaceCullPolicy {
    never,
    always,
    surface
};

struct ClippingPlaneGroupPlane {
    bool enabled;
    bool invertNormalDirection;
    ClippingPlaneGroupVisualization visualisation;
    ClippingPlaneGroupSlice slice;
    Transform transform;
};

struct ClippingPlaneGroupSlice {
    bool enabled = true;
    Vector color = {1.0, 0.0, 0.0};
    float thickness = 0.005;
};

struct ClippingPlaneGroupVisualization {
    bool planeEnabled;
    float planeWidth = 1.0;
    float planeHeight = 1.0;
    float planeAlpha = 0.1;
    Vector planeColor = {1.0, 0.0, 0.0};
    bool gridEnabled;
    float gridStep = 0.1;
    Vector gridColor = {1.0, 0.0, 0.0};
};
```

(continues on next page)

(continued from previous page)

```
};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<ClippingPlaneGroup, ClippingPlaneGroupProperties> properties;

};
```

Raptor_Configuration.dsm

Domain: ConfigurationExpression, ConfigurationRule

Defines configuration rules for product variants (e.g., car with/without sunroof).

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

struct ConfigurationExpressionProperties {
    ConfigurationExpressionOperationType operationType;
    optional<key<ConfigurationExpression>> leftExpressionKey;
    optional<key<ConfigurationExpression>> rightExpressionKey;
    string symbol;
};

concept ConfigurationExpression;

enum ConfigurationExpressionOperationType {
    defined,
    and,
    or,
    xor,
    not
};

club ConfigurationTargetSource;
membership ConfigurationTargetSource Model;
membership ConfigurationTargetSource Product;
membership ConfigurationTargetSource Overlay;

club ConfigurationTargetElement;
membership ConfigurationTargetElement GeometryLayer;
membership ConfigurationTargetElement AspectLayer;
membership ConfigurationTargetElement PositionLayer;
membership ConfigurationTargetElement EnvironmentLayer;
membership ConfigurationTargetElement LightingLayer;
membership ConfigurationTargetElement LightingLayerColorLayer;
membership ConfigurationTargetElement OverlayLayer;

struct ConfigurationTarget {
```

(continues on next page)

(continued from previous page)

```
    string name = "ConfigurationTarget";
    bool enabled;
    key<ConfigurationTargetSource> sourceKey;
    key<ConfigurationTargetElement> elementKey;
};

concept ConfigurationRule;
struct ConfigurationRuleProperties {
    string name = "ConfigurationRule";
    bool enabled = true;
    key<ConfigurationExpression> expressionKey;
    vector<ConfigurationTarget> targets;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<ConfigurationExpression, ConfigurationExpressionProperties> properties;
attachment<ConfigurationRule, ConfigurationRuleProperties> properties;

};
```

Raptor_Environment.dsm

Domain: Environment, EnvironmentGenerator, EnvironmentGeneratorHdrls, EnvironmentGeneratorLocal

Defines HDR environment lighting with support for HDRI files and local environment probes.

```
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Environment;
struct EnvironmentProperties {
    string name = "Environment";
    optional<key<Thumbnail>> thumbnailKey;
    optional<key<EnvironmentGenerator>> generatorKey;
    float gamma = 1.0;
    float saturation;
    float diffuseExposure;
    float diffuseColoration;
    float specularExposure;
    float backgroundExposure;
    float backgroundGamma = 1.0;
    Vector position;
    Transform defaultTransform;
    EnvironmentParallaxCorrection parallaxCorrection;
    key<TextureCube> diffuseCubeKey;
    vector<key<TextureCube>> specularCubeKeys;
    optional<key<Texture>> backgroundTextureKey;
    vector<EnvironmentLight> environmentLights;
```

(continues on next page)

(continued from previous page)

```
};

concept EnvironmentGenerator;

concept EnvironmentGeneratorHdrls is a EnvironmentGenerator;
struct EnvironmentGeneratorHdrlsProperties {
    bool immediateDataProcessing;
    int32 width;
    int32 height;
    blob_id data;
};

concept EnvironmentGeneratorLocal is a EnvironmentGenerator;
struct EnvironmentGeneratorLocalProperties {
    key<Product> productKey;
    float radius = 1.0;
    int32 resolution;
    set<string> surfaceTags;
    bool rebuildOnConfig;
};

concept EnvironmentLayer;
struct EnvironmentLayerProperties {
    string name = "EnvironmentLayer";
    bool enabled = true;
    map<key<Surface>, key<Environment>> environmentAssignments;
    map<key<Surface>, Transform> orientationAssignments;
};

struct EnvironmentLight {
    Vector direction;
    Vector color;
    float size;
};

struct EnvironmentParallaxCorrection {
    EnvironmentParallaxType parallaxType;
    Aabb aabb;
    Vector hemisphereCenter;
    float hemisphereRadius = 1.0;
    optional<key<Surface>> surfaceKey;
};

enum EnvironmentParallaxType {
    none,
    aabb,
    hemisphere,
    mesh
};

struct EnvironmentRenderProperties {
    bool hasEnvironmentOrientation;
```

(continues on next page)

(continued from previous page)

```
    Vector environmentOrientation;
    bool hasSunPosition;
    Vector sunPosition;
    bool overrideSun;
    EnvironmentSunProperties sunProperties;
};

struct EnvironmentSunProperties {
    bool enabled;
    Vector color;
    float intensity = 1.0;
    float shadowIntensity = 0.1;
    bool isSpecularEnabled;
    float specularIntensity;
    float lightmapIntensity;
};

struct SunAuthoringProperties {
    bool enabled;
    bool enabledShadowsInMirrors;
    Vector color = {1.0, 1.0, 1.0};
    float intensity = 1.0;
    float shadowIntensity;
    float lightmapIntensity = 1.0;
    SunShadowQuality shadowQuality;
    SunShadowSmoothness shadowSmoothness;
    bool isSpecularEnabled = true;
    float specularIntensity = 1.0;
    float northOrientation;
    SunOrientationType orientationType;
    SunAuthoringManualOrientation manualOrientation;
    SunAuthoringTimeAndLocationOrientation timeAndLocationOrientation;
};

struct SunAuthoringManualOrientation {
    float azimuth;
    float altitude = 30.0;
};

struct SunAuthoringTimeAndLocationOrientation {
    int32 year = 2000;
    uint8 month = 1;
    uint8 day = 1;
    uint8 hour = 12;
    uint8 minute;
    uint8 second;
    float timezone = 1.0;
    bool daylightSaving = true;
    int32 daylightSavingMinutes = 60;
    float latitude = 44.8386;
    float longitude = 0.5783;
};
```

(continues on next page)

(continued from previous page)

```
enum SunShadowQuality {
    veryLow,
    low,
    medium,
    fine,
    ultra
};

enum SunShadowSmoothness {
    none,
    weak,
    normal,
    fine,
    ultraFine,
    max
};

enum SunOrientationType {
    manual,
    timeAndLocation,
    extractedFromEnvironment
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Environment, EnvironmentProperties> properties;
attachment<EnvironmentGeneratorHdrls, EnvironmentGeneratorHdrlsProperties> properties;
attachment<EnvironmentGeneratorLocal, EnvironmentGeneratorLocalProperties> properties;
attachment<EnvironmentLayer, EnvironmentLayerProperties> properties;

};
```

Raptor_Iray.dsm

Domain: IraySettings, IrayMaterial, IrayMdlMaterial, IrayAxfMaterial

NVIDIA Iray path-tracing renderer integration with MDL materials and X-Rite AxF measured materials.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept IraySettings;
struct IraySettingsProperties {
    float maxQuality = 0.99;
    uint32 maxSamples = 512;
    float maxRenderTime = 3600.0;
    uint32 maxPathLength = 12;
```

(continues on next page)

(continued from previous page)

```
    float environmentExposure = 1.0;
    bool causticSampler;
    bool architecturalSampler;
    bool environmentMaterialEnabled;
    bool environmentMaterialUseAlternative;
    bool environmentBackgroundMode = true;
    IrayGround ground;
    IrayTonemapper tonemapper;
    IrayCameraEffects cameraEffects;
    bool firefliesFilteringEnabled;
    IrayDegrainFiltering degrain;
    IrayDenoiseFiltering denoise;
    IraySunSky sunSky;
};

struct IrayGround {
    bool enabled;
    float altitude;
    float shadowIntensity = 1.0;
    float scale = 1.5;
    float glossiness;
    Vector reflectivity = {1.0, 1.0, 1.0};
};

struct IrayTonemapper {
    bool enabled = true;
    IrayTonemappingMode mode;
    float burn = 0.2;
    float crush = 0.25;
    float ev = 7.0;
    float shutter = 0.125;
    float fNumber = 8.0;
    float filmIso = 100.0;
    float cm2Factor = 10.0;
    float saturation = 1.0;
    Vector whitePoint = {1.04287, 0.983863, 1.03358};
};

enum IrayTonemappingMode {
    standard,
    photographic
};

struct IrayCameraEffects {
    bool bloomEnabled;
    float bloomRadius = 0.01;
    float bloomThreshold = 0.9;
    float bloomBrightnessScale = 1.0;
    float vignetting;
};

struct IrayDegrainFiltering {
```

(continues on next page)

(continued from previous page)

```
    bool enabled;
    IrayDegrainMode mode;
    int32 radius = 3;
    float blurDifference = 0.05;
};

enum IrayDegrainMode {
    pixelClipping,
    smartMedian,
    smartAverage,
    limitedBlur,
    limitedAutoBlur
};

struct IrayDenoiseFiltering {
    bool enabled;
    uint32 minIterations = 8;
    uint32 maxMemory = 2048;
    bool denoiseAlpha;
};

struct IraySunSky {
    bool enabled;
    float multiplier = 0.09;
    Vector rgbUnitConversion = {0.000666667, 0.000666667, 0.000666667};
    float haze = 0.5;
    float redblueShift;
    float saturation = 0.5;
    float horizonHeight = 0.001;
    float horizonBlur = 0.1;
    Vector groundColor = {0.4, 0.4, 0.4};
    Vector nightColor;
    float sunDiskIntensity = 0.01;
    float sunDiskScale = 0.5;
    float sunGlowIntensity = 1.0;
    bool physicallyScaledSun = true;
};

struct IrayLight {
    float intensity = 1.0;
    float exponent = 1.0;
    bool useAsPortal;
    bool useRadiantExitance;
};

concept IrayMaterial;
struct IrayMaterialSettings {
    key<IrayMaterial> materialKey;
};

concept IrayMdlMaterial is a IrayMaterial;
struct IrayMdlMaterialProperties {
```

(continues on next page)

(continued from previous page)

```
    string name = "MdlMaterial";
    blob_id mdlData;
};

concept IrayAxfMaterial is a IrayMaterial;
struct IrayAxfMaterialProperties {
    blob_id axfData;
};

concept IrayStdMaterial is a IrayMaterial;
struct IrayStdMaterialProperties {
    float shadowIntensity;
};

concept IrayMatteMaterial is a IrayMaterial;
struct IrayMatteMaterialProperties {
    float shadowIntensity;
    Vector color;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<IraySettings, IraySettingsProperties> properties;

attachment<Light, IrayLight> iraySettings;
attachment<Material, IrayMaterialSettings> iraySettings;

attachment<IrayMdlMaterial, IrayMdlMaterialProperties> properties;
attachment<IrayAxfMaterial, IrayAxfMaterialProperties> properties;
attachment<IrayStdMaterial, IrayStdMaterialProperties> properties;
attachment<IrayMatteMaterial, IrayMatteMaterialProperties> properties;

};
```

Raptor_Kinematics.dsm

Domain: Kinematics, KinematicsNode, KinematicsNodeAxis, KinematicsNodeNull

Defines articulated motion for mechanical assemblies (doors, wheels, robotic arms).

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

club KinematicsElement;
membership KinematicsElement Surface;
membership KinematicsElement BezierPath;
membership KinematicsElement KinematicsNodeAxis;
membership KinematicsElement KinematicsNodeNull;
```

(continues on next page)

(continued from previous page)

```
membership KinematicsElement KinematicsNodeVector;

concept Kinematics;
struct KinematicsProperties {
    vector<key<KinematicsNode>> nodeKeys;
    map<key<KinematicsElement>, vector<key<KinematicsConstraint>>> constraints;
    map<key<KinematicsElement>, key<KinematicsElement>> parents;
};

concept KinematicsNode;

concept KinematicsNodeAxis is a KinematicsNode;
struct KinematicsNodeAxisProperties {
    string name = "KinematicsNodeAxis";
    float minAngle;
    float maxAngle;
};

concept KinematicsNodeNull is a KinematicsNode;
struct KinematicsNodeNullProperties {
    string name = "KinematicsNodeNull";
    vector<string> tags;
};

concept KinematicsNodeVector is a KinematicsNode;
struct KinematicsNodeVectorProperties {
    string name = "KinematicsNodeVector";
    float minDistance;
    float maxDistance;
};

concept KinematicsConstraint;

enum KinematicsConstraintAxis {
    x,
    y,
    z
};

concept KinematicsConstraintCopyOrientation is a KinematicsConstraint;
struct KinematicsConstraintCopyOrientationProperties {
    key<KinematicsElement> targetKey;
    Vector offset;
};

concept KinematicsConstraintCopyPosition is a KinematicsConstraint;
struct KinematicsConstraintCopyPositionProperties {
    key<KinematicsElement> targetKey;
    Vector offset;
};

concept KinematicsConstraintFollowPath is a KinematicsConstraint;
```

(continues on next page)

(continued from previous page)

```
struct KinematicsConstraintFollowPathProperties {
    key<KinematicsElement> targetKey;
    bool followCurve;
    KinematicsConstraintAxis followAimAxis;
    KinematicsConstraintAxis followUpAxis = .y;
    float curvilinearAbscissa = 0.0;
};

concept KinematicsConstraintLookAt is a KinematicsConstraint;
struct KinematicsConstraintLookAtProperties {
    key<KinematicsElement> targetKey;
    KinematicsConstraintAxis aimAxis;
    KinematicsConstraintAxis upAxis = .y;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Kinematics, KinematicsProperties> properties;
attachment<KinematicsNodeAxis, KinematicsNodeAxisProperties> properties;
attachment<KinematicsNodeNull, KinematicsNodeNullProperties> properties;
attachment<KinematicsNodeVector, KinematicsNodeVectorProperties> properties;
attachment<KinematicsConstraintCopyOrientation,␣
↪KinematicsConstraintCopyOrientationProperties> properties;
attachment<KinematicsConstraintCopyPosition,␣
↪KinematicsConstraintCopyPositionProperties> properties;
attachment<KinematicsConstraintFollowPath, KinematicsConstraintFollowPathProperties>␣
↪properties;
attachment<KinematicsConstraintLookAt, KinematicsConstraintLookAtProperties>␣
↪properties;

};
```

Raptor_Library.dsm

Domain: Library, CameraGroup, MaterialGroup, BackgroundGroup

Defines the **asset library system** - reusable presets organized in folders.

Library Groups

Each group is a folder concept:

- `Library`: root container for all asset groups
- `CameraGroup`: saved camera presets (viewpoints)
- `MaterialGroup`: material presets
- `BackgroundGroup`: background/environment presets
- `EnvironmentGroup`: HDR environment presets

- `LightGroup`: lighting presets
- `SensorGroup`: render settings presets
- `Folder`: generic subfolder for organization

Each group contains a `vector<key<...>>` of its items plus child folders, enabling nested organization like a file system.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Library;
struct LibraryProperties {
    key<CameraGroup> rootCameraGroupKey;
    key<MaterialGroup> rootMaterialGroupKey;
    key<Folder> rootTextureGroupKey;
    key<BackgroundGroup> rootBackgroundGroupKey;
    key<OverlayGroup> rootOverlayGroupKey;
    key<PostProcessGroup> rootPostprocessGroupKey;
    key<SensorGroup> rootSensorGroupKey;
    vector<key<ConfigurationRule>> configurationRuleKeys;
    map<key<Product>, vector<key<Camera>>> productCameras;
};

concept CameraGroup;
struct CameraGroupProperties {
    string name = "Camera Group";
    vector<key<CameraGroup>> groupKeys;
    vector<key<Camera>> cameraKeys;
};

concept MaterialGroup;
struct MaterialGroupProperties {
    string name = "Material Group";
    vector<key<MaterialGroup>> groupKeys;
    vector<key<Material>> materialKeys;
};

concept BackgroundGroup;
struct BackgroundGroupProperties {
    string name = "Background Group";
    vector<key<BackgroundGroup>> groupKeys;
    vector<key<Background>> backgroundKeys;
};

concept OverlayGroup;
struct OverlayGroupProperties {
    string name = "Overlay Group";
    vector<key<OverlayGroup>> groupKeys;
    vector<key<Overlay>> overlayKeys;
};

concept PostProcessGroup;
struct PostProcessGroupProperties {
    string name = "PostProcess Group";
```

(continues on next page)

(continued from previous page)

```
    vector<key<PostProcessGroup>> groupKeys;
    vector<key<PostProcess>> postProcessKeys;
};

concept SensorGroup;
struct SensorGroupProperties {
    string name = "Sensor Group";
    vector<key<SensorGroup>> groupKeys;
    vector<key<Sensor>> sensorKeys;
};

concept Folder;
struct FolderProperties {
    string name = "Folder";
    vector<key<Folder>> folderKeys;
    vector<uuid> entries;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Library, LibraryProperties> properties;

attachment<CameraGroup, CameraGroupProperties> properties;
attachment<MaterialGroup, MaterialGroupProperties> properties;
attachment<BackgroundGroup, BackgroundGroupProperties> properties;
attachment<OverlayGroup, OverlayGroupProperties> properties;
attachment<PostProcessGroup, PostProcessGroupProperties> properties;
attachment<SensorGroup, SensorGroupProperties> properties;

attachment<Folder, FolderProperties> properties;

};
```

Raptor_Lighting.dsm

Domain: LightingLayer, LightingLayerColorLayer, Light, LightSpot

Defines the **lighting system** for realistic illumination. Lights are organized in `LightingLayer` groups that can be toggled independently.

Light Hierarchy

Concept inheritance via `is a`:

- **`Light` (abstract base)**
  - `LightSpot` - cone-shaped spotlight with falloff
  - `LightOmni` - point light (omnidirectional)

  - `LightSun` - infinite directional light (outdoor scenes)
  - `LightSky` - ambient sky dome lighting
  - `LightAreaPlane` - soft rectangular light (studio softbox)
  - `LightAreaCylinder` - tubular light (fluorescent)
  - `LightAreaMesh` - light emitted from arbitrary geometry

Organization

- `LightingLayer`: groups lights that can be enabled/disabled together
- `LightingLayerColorLayer`: color grading per lighting layer
- Multiple LightingLayers per Model allow A/B lighting comparison

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept LightingLayer;
struct LightingLayerProperties {
    string name = "LightingLayer";
    bool enabled;
    float exposure = 1.0;
    float gamma = 1.0;
    vector<key<LightingLayerColorLayer>> colorLayerKeys;
    vector<key<Light>> lightKeys;
};

concept LightingLayerColorLayer;
struct LightingLayerColorLayerProperties {
    string name = "LightingLayerColorLayer";
    bool enabled;
    float intensity = 1.0;
    Vector color = {1.0, 1.0, 1.0};
};

concept Light;
struct LightProperties {
    string name;
    bool enabled = true;
    Vector color = {1.0, 1.0, 1.0};
    float intensity = 1.0;
    Vector position = {0.0, 0.1, 0.0};
    Vector orientation;
};

concept LightSpot is a Light;
struct LightSpotProperties {
    float diameter = 0.05;
    Vector target = {0.0, 0.0, 1.0};
    float falloff = 45.0;
    float hotSpot = 43.0;
    LightShadow shadow;
```

(continues on next page)

(continued from previous page)

```
    LightAttenuation attenuation;
    IESProfile iesProfile;
};

concept LightOmni is a Light;
struct LightOmniProperties {
    float diameter = 0.05;
    LightShadow shadow;
    LightAttenuation attenuation;
    IESProfile iesProfile;
};

concept LightSun is a Light;
struct LightSunProperties {
    float diameter = 0.05;
    LightShadow shadow;
};

concept LightSky is a Light;
struct LightSkyProperties {
    float topAngle = 0.05;
    float bottomAngle = 0.05;
    optional<key<Environment>> environmentKey;
    LightShadow shadow;
};

concept LightAreaPlane is a Light;
struct LightAreaPlaneProperties {
    float width = 1.0;
    float height = 1.0;
    LightShadow shadow;
    LightAttenuation attenuation;
    IESProfile iesProfile;
};

concept LightAreaCylinder is a Light;
struct LightAreaCylinderProperties {
    float diameter = 0.2;
    float length = 1.0;
    LightShadow shadow;
    LightAttenuation attenuation;
    IESProfile iesProfile;
};

concept LightAreaMesh is a Light;
struct LightAreaMeshProperties {
    key<Surface> surfaceKey;
    LightShadow shadow;
    LightAttenuation attenuation;
    IESProfile iesProfile;
};
```

(continues on next page)

(continued from previous page)

```
struct LightShadow {
    bool cast = true;
    float intensity = 0.1;
    ShadowIntegrity integrity = .normal;
};

struct LightAttenuation {
    LightAttenuationType attenuationType = .physical;
    bool bounded;
    float fullEffect = 2.0;
    float falloff = 2.1;
};

struct IESProfile {
    bool enabled;
    blob_id data;
    Vector orientation = {-90.0, 0.0, 0.0};
};

enum ShadowIntegrity {
    weak,
    normal,
    fine,
    ultraFine,
    max
};

enum LightAttenuationType {
    none,
    linearSlow,
    linearFast,
    quadraticSlow,
    quadraticFast,
    physical
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<LightingLayer, LightingLayerProperties> properties;
attachment<LightingLayerColorLayer, LightingLayerColorLayerProperties> properties;

attachment<Light, LightProperties> properties;
attachment<LightSpot, LightSpotProperties> properties;
attachment<LightOmni, LightOmniProperties> properties;
attachment<LightSun, LightSunProperties> properties;
attachment<LightSky, LightSkyProperties> properties;
attachment<LightAreaPlane, LightAreaPlaneProperties> properties;
attachment<LightAreaCylinder, LightAreaCylinderProperties> properties;
attachment<LightAreaMesh, LightAreaMeshProperties> properties;
```

(continues on next page)

(continued from previous page)

```
};
```

Raptor_Material.dsm

Domain: Material, MaterialEnvironment, MaterialMatte, MaterialMirror, MaterialMultilayer, MaterialStandard, MaterialAxfCpa2

The **heart of the rendering system** - defines how surfaces appear. This is the largest and most complex DSM file, showcasing concept hierarchy for material types.

Material Hierarchy

Concept inheritance via `is a`:

- **`Material` (abstract base)**
  - `MaterialStandard` - general-purpose PBR material (diffuse, specular, transparency)
  - `MaterialMultilayer` - stacked layers for complex surfaces (car paint, fabric)
  - `MaterialMirror` - perfect reflection with transparency
  - `MaterialMatte` - shadow catcher for compositing
  - `MaterialEnvironment` - HDR environment lighting
  - `MaterialSeam` - stitching/seam rendering for fabrics
  - `MaterialAxfCpa2` - X-Rite AxF measured material (real-world paint scans)

MaterialMultilayer Sublayers

- `MaterialMultilayerLayerDiffuse` - base color, ambient, color maps
- `MaterialMultilayerLayerSpecular` - highlights, roughness, fresnel
- `MaterialMultilayerLayerIllumination` - self-illumination, velvet effect

Note

The `is a` relationship enables polymorphism - a `key<Material>` can reference any material type, letting surfaces accept any material without knowing its specific type.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Material;

concept MaterialEnvironment is a Material;
struct MaterialEnvironmentProperties {
    string name = "MaterialEnvironment";
    optional<key<Thumbnail>> thumbnailKey;
    float intensity = 1.0;
```

(continues on next page)

(continued from previous page)

```
};

concept MaterialMatte is a Material;
struct MaterialMatteProperties {
    string name = "MaterialMatte";
    optional<key<Thumbnail>> thumbnailKey;
    float threshold;
    float offset;
    float outline;
    float contrast = 1.0;
};

concept MaterialMirror is a Material;
struct MaterialMirrorProperties {
    string name = "MaterialMirror";
    optional<key<Thumbnail>> thumbnailKey;
    Vector reflectionColor;
    bool isTransparent;
    Vector transparencyColor = {1.0, 1.0, 1.0};
    Vector interReflectionColor;
    MaterialMirrorOutboundSceneColor outboundSceneColor;
    bool reflectedSurfaceTagEnabled;
    string reflectedSurfaceTag;
};

enum MaterialMirrorOutboundSceneColor {
    black,
    background,
    environment
};

concept MaterialMultilayerLayer;

enum MaterialLabelMode {
    mix,
    mul,
    add
};

concept MaterialMultilayer is a Material;
struct MaterialMultilayerProperties {
    string name = "Material";
    optional<key<Thumbnail>> thumbnailKey;
    bool mipmapEnabled;
    bool layersUseRelief;
    bool isTransparent;
    vector<key<MaterialMultilayerLayer>> layerKeys;
    MaterialMultilayerRelief relief;
    TextureRepeatMode labelRepeatU;
    TextureRepeatMode labelRepeatV;
    MaterialLabelMode labelMode;
    float labelFactor = 1.0;
```

(continues on next page)

(continued from previous page)

```
    bool overrideRepeatForLabel = true;
};

struct MaterialMultilayerBump {
    bool enabled;
    float scale = 1.0;
    optional<key<BumpMap>> mapKey;
    Transform transform;
    TextureRepeatMode mapRepeatU = .repeat;
    TextureRepeatMode mapRepeatV = .repeat;
};

struct MaterialMultilayerRelief {
    bool enabled;
    float scale = 1.0;
    optional<key<BumpMap>> mapKey;
    optional<key<Texture>> maximumMipmapHeightMapKey;
    optional<key<Texture>> mipmapedHeightMapKey;
    Transform reliefMapTransform;
    TextureRepeatMode mapRepeatU = .repeat;
    TextureRepeatMode mapRepeatV = .repeat;
};

concept MaterialMultilayerLayerIllumination is a MaterialMultilayerLayer;
struct MaterialMultilayerLayerIlluminationProperties {
    string name = "MaterialMultilayerLayerIllumination";
    bool enabled = true;
    float intensity;
    Vector color;
    float inShadow;
    float velvetFactor;
    optional<key<Texture>> velvetMapKey;
    bool velvetMapEnabled;
    bool velvetMapModulateEnabled;
    optional<key<Texture>> modulateMapKey;
    bool modulateMapEnabled;
    TextureRepeatMode modulateMapRepeatU = .repeat;
    TextureRepeatMode modulateMapRepeatV = .repeat;
    Transform modulateMapTransform;
    bool allowTextureRepeat;
    MaterialMultilayerBump bump;
    bool useRelief;
};

concept MaterialMultilayerLayerDiffuse is a MaterialMultilayerLayer;
struct MaterialMultilayerLayerDiffuseProperties {
    string name = "MaterialMultilayerLayerDiffuse";
    bool enabled = true;
    float intensity = 1.0;
    Vector color;
    Vector ambientColor;
    optional<key<Texture>> colorMapKey;
```

(continues on next page)

(continued from previous page)

```
    bool colorMapEnabled = false;
    TextureRepeatMode colorMapRepeatU = .repeat;
    TextureRepeatMode colorMapRepeatV = .repeat;
    Transform colorMapTransform;
    float alphaModulator = 1.0;
    optional<key<Texture>> alphaMapKey;
    TextureRepeatMode alphaMapRepeatU = .repeat;
    TextureRepeatMode alphaMapRepeatV = .repeat;
    Transform alphaMapTransform;
    optional<key<Texture>> filterMapKey;
    bool filterMapEnabled;
    MaterialMultilayerBump bump;
    bool useRelief;
};

concept MaterialMultilayerLayerSpecular is a MaterialMultilayerLayer;
struct MaterialMultilayerLayerSpecularProperties {
    string name = "MaterialMultilayerLayerSpecular";
    bool enabled = true;
    float roughness;
    float intensity = 1.0;
    float inShadow;
    bool fresnelEnabled = false;
    float fresnelRefraction;
    float fresnelExtinction;
    bool transmissionAttenuationEnabled;
    optional<key<Texture>> modulateMapKey;
    bool modulateMapEnabled;
    TextureRepeatMode modulateMapRepeatU = .repeat;
    TextureRepeatMode modulateMapRepeatV = .repeat;
    Transform modulateMapTransform;
    optional<key<Texture>> roughnessMapKey;
    bool roughnessMapEnabled;
    TextureRepeatMode roughnessMapRepeatU = .repeat;
    TextureRepeatMode roughnessMapRepeatV = .repeat;
    Transform roughnessMapTransform;
    Vector filter;
    Vector diffuseFilter;
    optional<key<Texture>> filterMapKey;
    bool filterMapEnabled;
    MaterialMultilayerBump bump;
    bool useRelief;
};

concept MaterialSeam is a Material;
struct MaterialSeamProperties {
    string name = "MaterialSeam";
    optional<key<Thumbnail>> thumbnailKey;
    Vector diffuseColor;
    Vector ambientColor;
    float diffuseIntensity = 1.0;
    optional<key<Texture>> diffuseMapKey;
```

(continues on next page)

(continued from previous page)

```
    bool diffuseMapEnabled;
    optional<key<Texture>> seamMapKey;
    bool seamMapEnabled;
    optional<key<BumpMap>> seamBumpMapKey;
    float specularRoughness;
    float specularIntensity;
    Vector specularFilter;
    Vector specularDiffuseFilter;
    optional<key<BumpMap>> pleatMapKey;
    bool pleatMapEnabled;
    float pleatMapBumpScale;
    bool keepAspectRatio;
    bool mipmapEnabled;
    Transform transformation;
    bool bumpDiffuseEnabled;
    float bumpDiffuseScale;
    bool bumpSpecularEnabled;
    float bumpSpecularScale;
};

concept MaterialStandard is a Material;

enum MaterialStandardType {
    diffuse,
    diffuseSpecular,
    transparent
};

struct MaterialStandardProperties {
    string name = "MaterialStandard";
    optional<key<Thumbnail>> thumbnailKey;
    MaterialStandardType materialType = .diffuseSpecular;
    Vector diffuseColor;
    Vector ambientColor;
    Vector illuminationColor;
    float diffuseIntensity;
    float illuminationIntensity;
    optional<key<Texture>> diffuseMapKey;
    bool diffuseMapEnabled;
    TextureRepeatMode diffuseMapRepeatU = .repeat;
    TextureRepeatMode diffuseMapRepeatV = .repeat;
    Transform diffuseMapTransform;
    float alphaModulator;
    optional<key<Texture>> alphaMapKey;
    Transform alphaMapTransform;
    TextureRepeatMode alphaMapRepeatU = .repeat;
    TextureRepeatMode alphaMapRepeatV = .repeat;
    optional<key<Texture>> diffuseFilterMapKey;
    bool diffuseFilterMapEnabled;
    optional<key<BumpMap>> bumpMapKey;
    Transform bumpMapTransform;
    TextureRepeatMode bumpMapRepeatU = .repeat;
```

(continues on next page)

(continued from previous page)

```
    TextureRepeatMode bumpMapRepeatV = .repeat;
    bool bumpDiffuseEnabled;
    float bumpDiffuseScale;
    bool bumpSpecularEnabled;
    float bumpSpecularScale;
    optional<key<Texture>> maximumMipmapHeightMapKey;
    optional<key<Texture>> mipmapedHeightMapKey;
    bool reliefBumpEnabled;
    float reliefBumpScale;
    float specularRoughness;
    float specularIntensity;
    float specularInShadow;
    bool fresnelEnabled;
    float fresnelRefraction;
    float fresnelExtinction;
    bool diffuseAttenuationEnabled;
    bool velvetEnabled;
    float velvetFactor;
    optional<key<Texture>> velvetMapKey;
    bool velvetMapEnabled;
    bool velvetMapModulateEnabled;
    optional<key<Texture>> specularModulateMapKey;
    bool specularModulateMapEnabled;
    TextureRepeatMode specularModulateMapRepeatU = .repeat;
    TextureRepeatMode specularModulateMapRepeatV = .repeat;
    Transform specularModulateMapTransform;
    optional<key<Texture>> roughnessMapKey;
    bool roughnessMapEnabled;
    TextureRepeatMode roughnessMapRepeatU = .repeat;
    TextureRepeatMode roughnessMapRepeatV = .repeat;
    Transform roughnessMapTransform;
    bool roughnessMapIsGloss;
    Vector specularFilter;
    Vector specularDiffuseFilter;
    bool transformationLink;
    bool mipmapEnabled;
    TextureRepeatMode labelRepeatU;
    TextureRepeatMode labelRepeatV;
    MaterialLabelMode labelMode;
    float labelFactor = 1.0;
};

concept MaterialAxfCpa2 is a Material;
struct MaterialAxfCpa2Properties {
    string name = "MaterialAxfCpa2";
    optional<key<Thumbnail>> thumbnailKey;
    float diffuse = 1.0;
    vector<float> coeffs;
    vector<float> f0s;
    vector<float> spreads;
    float ior = 1.0;
    bool refraction = true;
```

(continues on next page)

(continued from previous page)

```
    optional<key<BumpMap>> clearCoatBumpMapKey;
    int32 numThetaF;
    int32 numThetaI;
    int32 maxThetaI;
    vector<int32> sliceLUT;
    Transform clearCoatTransform;
    Transform flakesTransform;
    optional<key<Texture>> colorsKey;
    optional<key<TextureArray>> flakesKey;
    float hueShift;
    float saturation = 1.0;
    float contrast = 1.0;
    float exposure = 1.0;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<MaterialEnvironment, MaterialEnvironmentProperties> properties;
attachment<MaterialMatte, MaterialMatteProperties> properties;
attachment<MaterialMirror, MaterialMirrorProperties> properties;
attachment<MaterialMultilayer, MaterialMultilayerProperties> properties;
attachment<MaterialMultilayerLayerIllumination,␣
↪MaterialMultilayerLayerIlluminationProperties> properties;
attachment<MaterialMultilayerLayerDiffuse, MaterialMultilayerLayerDiffuseProperties>␣
↪properties;
attachment<MaterialMultilayerLayerSpecular, MaterialMultilayerLayerSpecularProperties>
↪ properties;
attachment<MaterialSeam, MaterialSeamProperties> properties;
attachment<MaterialStandard, MaterialStandardProperties> properties;
attachment<MaterialAxfCpa2, MaterialAxfCpa2Properties> properties;

};
```

Raptor_Math.dsm

Domain: Core mathematical types

Foundational structs used throughout the Raptor model.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

struct Vector {
    float x;
    float y;
    float z;
};
```

(continues on next page)

(continued from previous page)

```
struct Aabb {
    Vector min;
    Vector max;
};

struct Transform {
    Vector translation;
    Vector orientation;
    Vector scaling = {1.0, 1.0, 1.0};
};

struct Plane {
    Vector normal;
    float q;
};

struct PointOfView {
    Vector target;
    Vector eye = {2.0, 2.0, 2.0};
    Vector up = {0.0, 1.0, 0.0};
};

};
```

Raptor_Mesh.dsm

Domain: Mesh, MeshAnimation, Thumbnail

Defines 3D geometry data with support for animation and preview thumbnails.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Mesh;
struct MeshProperties {
    Aabb aabb;
    int32 triangleCount;
    int32 vertexCount;
    blob_id blob_indices;
    blob_id blob_positions;
    blob_id blob_normals;
    blob_id blob_tangents;
    blob_id blob_binormals;
    blob_id blob_lightmapUvs;
    map<int8, blob_id> blob_uvs;
    map<key<LightingLayer>, blob_id> blob_directions;
    blob_id blob_animation;
};

concept MeshAnimation;
struct MeshAnimationProperties {
```

(continues on next page)

(continued from previous page)

```
    int32 vertexCount;
    int32 defaultFrame;
    vector<MeshAnimationFrame> frames;
};

struct MeshAnimationFrame {
    Aabb aabb;
    blob_id blob_positions;
    blob_id blob_normals;
    blob_id blob_tangents;
    blob_id blob_binormals;
};

concept Thumbnail;
struct ThumbnailProperties {
    string name = "Thumbnail";
    vector<Mipmap> mipmaps;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Mesh, MeshProperties> properties;
attachment<MeshAnimation, MeshAnimationProperties> properties;
attachment<Thumbnail, ThumbnailProperties> properties;

};
```

Raptor_Model.dsm

Domain: Model, GeometryLayer, PositionLayer

Defines the **scene graph structure** - how 3D geometry is organized hierarchically.

Concepts

- `Model`: root container for a complete 3D model (e.g., a car)
- `GeometryLayer`: hierarchical node containing Surfaces and child GeometryLayers (tree structure)
- `PositionLayer`: transformation overrides for kinematics (different poses/configurations)

Key Relationships

- Model references LightingLayers, Kinematics, PositionLayers, BezierPaths
- GeometryLayer uses tree structure via `childrenKeys`, contains `surfaceKeys`
- PositionLayer stores `localToPivot` and `pivotToParent` transforms per KinematicsElement

This hierarchical model enables part visibility toggling, level-of-detail, and configuration variants (e.g., car with/without spoiler).

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Model;
struct ModelProperties {
    string name = "Model";
    vector<key<LightingLayer>> lightingLayerKeys;
    key<GeometryLayer> rootGeometryLayerKey;
    key<Kinematics> kinematicsKey;
    vector<key<PositionLayer>> positionLayerKeys;
    vector<key<BezierPath>> bezierPathKeys;
};

concept GeometryLayer;
struct GeometryLayerProperties {
    string name = "GeometryLayer";
    bool enabled = true;
    vector<key<GeometryLayer>> childrenKeys;
    vector<key<Surface>> surfaceKeys;
};

concept PositionLayer;
struct PositionLayerProperties {
    string name = "PositionLayer";
    bool enabled;
    map<key<KinematicsElement>, Transform> localToPivot;
    map<key<KinematicsElement>, Transform> pivotToParent;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Model, ModelProperties> properties;
attachment<GeometryLayer, GeometryLayerProperties> properties;
attachment<PositionLayer, PositionLayerProperties> properties;

};
```

Raptor_Product.dsm

Domain: Product, AspectLayer

Defines products with material variants (AspectLayers) and configuration options.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Product;
struct ProductProperties {
```

(continues on next page)

(continued from previous page)

```
    string name = "Product";
    optional<key<Thumbnail>> thumbnailKey;
    key<Model> modelKey;
    vector<key<AspectLayer>> aspectLayerKeys;
    bool materialsHideLabelsBelow;
    map<key<Surface>, SurfaceRenderProperties> surfaceRenderProperties;
    vector<key<EnvironmentLayer>> environmentLayerKeys;
    map<key<Environment>, EnvironmentRenderProperties> environmentRenderProperties;
    map<string, set<string>> configurationBookmarks;
    set<string> configurationDefines;
    bool ignoreBackfaceCull;
    bool environmentLinkedToDiffuse = true;
    SunAuthoringProperties sunAuthoring;
    SSAOProperties ssao;
};

enum BackfaceCullMode {
    show,
    default,
    hide
};

struct SurfaceRenderProperties {
    bool isHidden;
    BackfaceCullMode backfaceCullMode = .default;
    bool transparencyDepthWrite;
};

struct MaterialAssignment {
    key<Material> materialKey;
    Transform transform;
    int8 uvSet;
    Plane mirrorPlane;
};

struct LabelAssignment {
    string name = "LabelAssignment";
    bool enabled = true;
    MaterialAssignment materialAssignment;
};

concept AspectLayer;
struct AspectLayerProperties {
    string name = "AspectLayer";
    bool enabled;
    map<key<Surface>, MaterialAssignment> materialAssignments;
    map<key<Surface>, vector<LabelAssignment>> labelAssignments;
};

struct SSAOProperties {
    bool enabled;
    bool lightmaps = true;
```

(continues on next page)

(continued from previous page)

```
    bool transparentSurfaces;
    float radius = 0.05;
    float intensity = 1.0;
    float bias = 8.0;
    uint16 steps = 4;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Product, ProductProperties> properties;
attachment<AspectLayer, AspectLayerProperties> properties;

};
```

Raptor_Sensor.dsm

Domain: Sensor, Background, Overlay, OverlayLayer, PostProcess

Defines render output settings including backgrounds, overlays, and post-processing effects.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Sensor;

enum SensorBackgroundType {
    none,
    gradient,
    environment
};

struct SensorProperties {
    string name = "Sensor";
    optional<key<Thumbnail>> thumbnailKey;
    bool isometric;
    bool dynamicAspectRatio;
    float width = 0.024;
    float height = 0.036;
    optional<key<Background>> backgroundKey;
    SensorBackgroundType backgroundType;
    optional<key<Overlay>> overlayKey;
    bool overlayVisibility;
    optional<key<PostProcess>> postprocessKey;
    bool postprocessVisibility;
};

concept Background;
struct BackgroundProperties {
```

(continues on next page)

(continued from previous page)

```
    string name = "Background";
    optional<key<Thumbnail>> thumbnailKey;
    Vector gradientStart = {0.455, 0.455, 0.455};
    Vector gradientEnd = {0.455, 0.455, 0.455};
    bool gradientActivated = true;
    float gradientOrientationAngle;
    bool preserveTextureAspect = true;
    optional<key<Texture>> textureKey;
    bool textureEnabled;
    Transform textureTransform;
};

concept Overlay;
struct OverlayProperties {
    string name = "Overlay";
    optional<key<Thumbnail>> thumbnailKey;
    float alpha = 1.0;
    vector<key<OverlayLayer>> layerKeys;
};

concept OverlayLayer;
struct OverlayLayerProperties {
    string name = "OverlayLayer";
    bool enabled;
    OverlayLayerType layerType = .sprite;
    OverlayLayerSizeType layerSize;
    float width = 0.25;
    float height = 0.25;
    OverlayLayerLengthUnit widthUnit;
    OverlayLayerLengthUnit heightUnit;
    bool constrainedRotation = true;
    OverlayLayerVerticalAlignment verticalAlignment = .bottom;
    OverlayLayerHorizontalAlignment horizontalAlignment;
    Transform transform;
    OverlayLayerLengthUnit offsetUUnit;
    OverlayLayerLengthUnit offsetVUnit;
    Vector gradientColorStart = {1.0, 1.0, 1.0};
    Vector gradientColorEnd = {1.0, 1.0, 1.0};
    float gradientAlphaStart = 1.0;
    float gradientAlphaEnd = 1.0;
    bool gradientFlipVertically;
    bool gradientFlipHorizontally;
    optional<key<Texture>> textureKey;
    bool textureEnabled;
    Transform textureTransform;
};

enum OverlayLayerHorizontalAlignment {
    left,
    middle,
    right
};
```

(continues on next page)

(continued from previous page)

```
enum OverlayLayerLengthUnit {
    pixel,
    millimeters,
    centimeters,
    inches,
    relativeToX,
    relativeToY
};

enum OverlayLayerSizeType {
    texture,
    screen,
    userDefined
};

enum OverlayLayerType {
    sticker,
    sprite
};

enum OverlayLayerVerticalAlignment {
    top,
    middle,
    bottom
};

concept PostProcess;
struct PostProcessProperties {
    string name = "PostProcess";
    optional<key<Thumbnail>> thumbnailKey;
    bool applyToBackground = true;
    bool applyToOverlay = true;
    optional<uint32> soloEffectIndex;
    vector<PostProcessEffect> effects;
};

enum PostProcessDataType {
    bool,
    int,
    float,
    color,
    length,
    texture,
    levels,
    cameraResponseEnum
};

struct PostProcessEffect {
    string name = "PostProcessEffect";
    PostProcessEffectType effectType;
    bool enabled;
```

(continues on next page)

(continued from previous page)

```
    vector<PostProcessEffectParameter> parameters;
};

struct PostProcessEffectParameter {
    string name = "PostProcessEffectParameter";
    int32 parameterLength = 1;
    PostProcessLengthUnit lengthUnit = .none;
    PostProcessDataType dataType = .float;
    string parameterValue = "0/";
};

enum PostProcessEffectType {
    negative,
    blackAndWhite,
    sepia,
    grayscale,
    colorFilter,
    blurHorizontal,
    blurVertical,
    grainGeneratorPerlin,
    grainGeneratorMd4,
    edgeDetector,
    combine,
    erodeHorizontal,
    erodeVertical,
    erode,
    dilateHorizontal,
    dilateVertical,
    dilate,
    adjustColor,
    automaticToneMapping,
    blur,
    grain,
    handDrawing,
    store,
    restore,
    get3DImage,
    glowThresholder,
    glowCombiner,
    glow,
    sharpenCombiner,
    sharpen,
    multiplyAdd,
    bloom,
    levels,
    bloomCombiner,
    reinhardToneMapping,
    dragoToneMapping,
    bloomThresholder,
    vignetting,
    cameraResponse
};
```

(continues on next page)

(continued from previous page)

```
enum PostProcessLengthUnit {
    pixel,
    millimeter,
    none,
    relativeToX,
    relativeToY,
    relativeToDefault
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Sensor, SensorProperties> properties;
attachment<Background, BackgroundProperties> properties;
attachment<Overlay, OverlayProperties> properties;
attachment<OverlayLayer, OverlayLayerProperties> properties;
attachment<PostProcess, PostProcessProperties> properties;

};
```

Raptor_Surface.dsm

Domain: Surface

Defines 3D surfaces (geometry instances) with mesh references and rendering properties.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Surface;
struct SurfaceProperties {
    string name = "Surface";
    map<key<LightingLayer>, key<Texture>> lightmaps;
    key<Mesh> meshKey;
    Vector color = {1.0, 1.0, 1.0};
    set<string> tags;
    SurfaceBillboardMode billboardMode;
    optional<key<MeshAnimation>> meshAnimationKey;
    SurfaceMirrorPlane mirrorPlane;
};

enum SurfaceBillboardMode {
    none,
    rotateY,
    rotateXy
};

struct SurfaceMirrorPlane {
```

(continues on next page)

(continued from previous page)

```
    Vector position;
    Vector normal = {0.0, 1.0, 0.0};
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

attachment<Surface, SurfaceProperties> properties;

};
```

Raptor_Texture.dsm

Domain: BumpMap, Texture, TextureCube, TextureArray, Video

Defines **texture assets** used by materials for color, bump, and environment mapping.

Texture Types

- `Texture`: 2D image with mipmaps (diffuse, specular, alpha maps)
- `TextureCube`: 6-face cubemap for environment reflections (xPos, xNeg, yPos, yNeg, zPos, zNeg)
- `TextureArray`: stack of textures for layered effects (car paint flakes)
- `BumpMap`: normal/height maps for surface detail
- `Video`: animated texture source

Key Structures

- `Mipmap`: single mip level with `width`, `height`, `format`, and `blob_id` referencing pixel data
- `TextureRepeatMode`: enum for UV wrapping (repeat, clamp, mirror)

Note

Blob pattern: Actual pixel data is stored in `blob_id` references, not inline - enabling efficient deduplication and streaming of large textures.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

struct Mipmap {
    int32 width;
    int32 height;
    blob_id blob_image;
};
```

(continues on next page)

(continued from previous page)

```
concept BumpMap;
struct BumpMapProperties {
    string name = "BumpMap";
    optional<key<Thumbnail>> thumbnailKey;
    vector<Mipmap> mipmaps;
};

concept Texture;
struct TextureProperties {
    string name = "Texture";
    optional<key<Thumbnail>> thumbnailKey;
    vector<Mipmap> mipmaps;
    optional<key<Video>> videoKey;
};

concept TextureCube;
struct TextureCubeProperties {
    string name = "TextureCube";
    vector<Mipmap> xPosMipmaps;
    vector<Mipmap> xNegMipmaps;
    vector<Mipmap> yPosMipmaps;
    vector<Mipmap> yNegMipmaps;
    vector<Mipmap> zPosMipmaps;
    vector<Mipmap> zNegMipmaps;
};

concept TextureArray;
struct TextureArrayProperties {
    string name = "TextureArray";
    optional<key<Thumbnail>> thumbnailKey;
    vector<vector<Mipmap>> textures;
};

enum TextureRepeatMode {
    clamp,
    repeat,
    mirroredRepeat
};

concept Video;
struct VideoProperties {
    string name = "Video";
    optional<key<Thumbnail>> thumbnailKey;
    float duration;
    blob_id blob_video;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {
```

(continues on next page)

(continued from previous page)

```
attachment<BumpMap, BumpMapProperties> properties;
attachment<Texture, TextureProperties> properties;
attachment<TextureCube, TextureCubeProperties> properties;
attachment<TextureArray, TextureArrayProperties> properties;
attachment<Video, VideoProperties> properties;

};
```

Raptor_Timeline.dsm

Domain: Timeline, TimelineClip, TimelineClipCameraBezierPath, TimelineClipCameraBookmark

Defines the **animation system** for creating cinematic sequences. Timelines contain clips that animate cameras, configurations, and properties over time.

Timeline Structure

- `Timeline`: container with tracks (TimelineTrack) and triggers
- `TimelineTrack`: ordered sequence of clips on a single channel
- `TimelineClip`: base concept for all animation clips

Clip Types

TimelineClip hierarchy:

- `TimelineClipCameraBezierPath`: animate camera along a Bezier spline
- `TimelineClipCameraBookmark`: jump between saved camera positions
- `TimelineClipCameraKamFile`: import external camera animation
- `TimelineClipConfiguration`: switch product configurations over time
- `TimelineClipVideo`: play video textures
- `TimelineClipChannelCurve`: animate any property with keyframes and curves
- `TimelineClipChannelBaked`: pre-computed animation data
- `TimelineClipChannelSimple`: linear interpolation between two values

Triggers

- `TimelineTriggerKey`: trigger actions at specific frames (keyboard shortcuts)
- `TimelineTriggerSurface`: trigger when clicking surfaces (interactive presentations)

This enables creating product configurators, turntable animations, and interactive experiences.

```
// Types
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {

concept Timeline;
struct TimelineProperties {
```

(continues on next page)

(continued from previous page)

```
    string name = "Timeline";
    float timeStart;
    float timeEnd;
    vector<TimelineTrack> tracks;
};

club TimelineAnimation;
membership TimelineAnimation Timeline;
membership TimelineAnimation TimelineClipCameraBezierPath;
membership TimelineAnimation TimelineClipCameraBookmark;
membership TimelineAnimation TimelineClipCameraKamFile;
membership TimelineAnimation TimelineClipChannelBaked;
membership TimelineAnimation TimelineClipChannelCurve;
membership TimelineAnimation TimelineClipChannelSimple;
membership TimelineAnimation TimelineClipConfiguration;
membership TimelineAnimation TimelineClipProduct;
membership TimelineAnimation TimelineClipVideo;

concept TimelineClip;

concept TimelineClipCameraBezierPath is a TimelineClip;
struct TimelineClipCameraBezierPathProperties {
    string name = "TimelineClipCameraBezierPath";
    float duration;
    TimelineClipCameraBezierPathPosition positionType = .fixed;
    TimelineClipCameraBezierPathDirection directionType = .fixedDirection;
    optional<key<BezierPath>> bezierPathUsedForPositionKey;
    optional<key<BezierPath>> bezierPathUsedForDirectionKey;
    bool invertBezierPathUsedForPosition;
    bool invertBezierPathUsedForDirection;
    optional<key<KinematicsNodeNull>> nullUsedForPositionKey;
    optional<key<KinematicsNodeNull>> nullUsedForDirectionKey;
    Vector fixedPositionFrom = {1.0, 1.0, 1.0};
    Vector fixedPositionTo;
    Vector fixedDirection = {0.0, 0.0, 1.0};
    Vector fixedDirectionUp = {0.0, 1.0, 0.0};
    Vector fixedTargetPosition;
};

enum TimelineClipCameraBezierPathDirection {
    followBezierPath,
    followBezierPathPosition,
    fixedDirection,
    fixedPosition,
    followNull
};

enum TimelineClipCameraBezierPathPosition {
    followBezierPath,
    fixed,
    followNull
};
```

(continues on next page)

(continued from previous page)

```
concept TimelineClipCameraBookmark is a TimelineClip;
struct TimelineClipCameraBookmarkProperties {
    string name = "TimelineClip";
    float duration;
    float fps;
    float sleepAddCst;
    float sleepMulCst;
    float durationAddCst;
    float durationMulCst;
    bool closedPath;
    vector<TimelineClipCameraBookmarkBookmark> bookmarks;
};

struct TimelineClipCameraBookmarkBookmark {
    string name = "Bookmark";
    bool enabled;
    TimelineClipCameraProperties cameraProperties;
    TimelineClipCameraBookmarkTransition transition = .linear;
    float transitionSleep;
    float transitionSmoothness;
    float transitionDuration = 1.0;
    float transitionTurn = 1.0;
    float transitionTurnDuration = 1.0;
};

enum TimelineClipCameraBookmarkTransition {
    linear,
    jump,
    orbit,
    head,
    spline
};

struct TimelineClipCameraFieldOfView {
    CameraFovType fovType;
    float fov = 0.7853981634;
};

concept TimelineClipCameraKamFile is a TimelineClip;
struct TimelineClipCameraKamFileProperties {
    string name = "TimelineClip";
    float duration;
    key<TimelineKamFile> kamFileKey;
};

struct TimelineClipCameraProperties {
    TimelineClipCameraFieldOfView fieldOfView;
    PointOfView pointOfView;
};

concept TimelineClipChannelBaked is a TimelineClip;
```

(continues on next page)

(continued from previous page)

```
struct TimelineClipChannelBakedProperties {
    string name = "TimelineClip";
    float duration;
    blob_id blob_transforms;
    uuid target;
    uuid subTarget;
    uuid elementType;
    uuid element;
};

concept TimelineClipChannelCurve is a TimelineClip;
struct TimelineClipChannelCurveProperties {
    string name = "TimelineClip";
    float duration;
    vector<TimelineClipChannelCurveChannel> channels;
};

struct TimelineClipChannelCurveChannel {
    Vector color;
    vector<TimelineClipChannelCurveKeyframe> keyframes;
    uuid target;
    uuid subTarget;
    uuid elementType;
    uuid element;
};

struct TimelineClipChannelCurveKeyframe {
    float time;
    float value;
    Vector leftTangent;
    Vector rightTangent;
    TimelineClipChannelCurveTangent rightTangentType = .linear;
};

enum TimelineClipChannelCurveTangent {
    bezier,
    linear,
    step
};

concept TimelineClipChannelSimple is a TimelineClip;
struct TimelineClipChannelSimpleProperties {
    string name = "TimelineClip";
    float duration;
    float valueStart;
    float valueEnd;
    TimelineClipChannelSimpleEasing easing = .linear;
    uuid target;
    uuid subTarget;
    uuid elementType;
    uuid element;
};
```

(continues on next page)

(continued from previous page)

```
enum TimelineClipChannelSimpleEasing {
    linear,
    inQuad,
    outQuad,
    inOutQuad
};

concept TimelineClipConfiguration is a TimelineClip;
struct TimelineClipConfigurationProperties {
    string name = "TimelineClip";
    vector<TimelineClipConfigurationParameter> parameters;
};

struct TimelineClipConfigurationParameter {
    string name = "TimelineClipConfigurationParameter";
    string value;
    bool isBinary;
};

concept TimelineClipProduct is a TimelineClip;
struct TimelineClipProductProperties {
    string name = "TimelineClip";
    key<Product> productKey;
};

concept TimelineClipVideo is a TimelineClip;
struct TimelineClipVideoProperties {
    string name = "TimelineClip";
    float duration;
    key<Video> videoKey;
};

concept TimelineData;
struct TimelineDataProperties {
    vector<key<Timeline>> timelineKeys;
    vector<key<TimelineClip>> clipKeys;
    vector<key<TimelineTrigger>> triggerKeys;
};

concept TimelineKamFile;
struct TimelineKamFileProperties {
    float fps = 30.0;
    bool ignoreFov;
    bool swapFov;
    blob_id blob_frames;
};

enum TimelineLoopType {
    none,
    repeated,
    incremented
```

(continues on next page)

(continued from previous page)

```
};

enum TimelineTrackType {
    product,
    configuration,
    camera,
    channels,
    video
};

struct TimelineTrack {
    TimelineTrackType trackType = .channels;
    vector<TimelineTrackEntry> entries;
};

struct TimelineTrackEntry {
    key<TimelineClip> clipKey;
    bool isReversed;
    TimelineLoopType loopType;
    float time;
};

concept TimelineTrigger;

struct TimelineTriggerAnimation {
    TimelineTriggerPlayMode playMode;
    key<TimelineAnimation> animationKey;
};

concept TimelineTriggerKey is a TimelineTrigger;
struct TimelineTriggerKeyProperties {
    int64 key;
    vector<TimelineTriggerAnimation> animations;
};

enum TimelineTriggerPlayMode {
    continued,
    reset,
    invertWithPause,
    invertWithoutPause
};

concept TimelineTriggerSurface is a TimelineTrigger;
struct TimelineTriggerSurfaceProperties {
    key<Surface> surfaceKey;
    vector<TimelineTriggerAnimation> animations;
};

};

// Attachments
namespace Raptor {f2d9ea90-2adc-4e9a-a2bf-02288281747d} {
```

(continues on next page)

(continued from previous page)

```
attachment<Timeline, TimelineProperties> properties;
attachment<TimelineClipCameraBezierPath, TimelineClipCameraBezierPathProperties>␣
↪properties;
attachment<TimelineClipCameraBookmark, TimelineClipCameraBookmarkProperties>␣
↪properties;
attachment<TimelineClipCameraKamFile, TimelineClipCameraKamFileProperties> properties;
attachment<TimelineClipChannelBaked, TimelineClipChannelBakedProperties> properties;
attachment<TimelineClipChannelCurve, TimelineClipChannelCurveProperties> properties;
attachment<TimelineClipChannelSimple, TimelineClipChannelSimpleProperties> properties;
attachment<TimelineClipConfiguration, TimelineClipConfigurationProperties> properties;
attachment<TimelineClipProduct, TimelineClipProductProperties> properties;
attachment<TimelineClipVideo, TimelineClipVideoProperties> properties;
attachment<TimelineData, TimelineDataProperties> properties;
attachment<TimelineKamFile, TimelineKamFileProperties> properties;
attachment<TimelineTriggerKey, TimelineTriggerKeyProperties> properties;
attachment<TimelineTriggerSurface, TimelineTriggerSurfaceProperties> properties;

};
```

3.2 Python Guide

This manual covers the **dsviper** Python API — the Python binding of the Viper runtime. You will learn how to use the type and value system, work with databases, and leverage the full power of Viper from Python.

3.2.1 The Metadata Everywhere Principle

Viper carries its type information as runtime metadata, and every subsystem draws on the same source: types, values, functions, serialization, database persistence, and RPC. You describe your data once — either through the static API generated from a DSM model, or directly through the dynamic API — and the runtime handles conversion, validation, and cross-language interoperability for you. This is why Python natives (`list`, `dict`, `tuple`) can be passed directly to typed `dsviper` functions: the metadata drives the bridge, so there is nothing to register and nothing to glue by hand.

Installation

This chapter covers installing and verifying the dsviper Python module.

Prerequisites

- Python 3.14+
- pip package manager

Installing from PyPI

The simplest install is directly from PyPI:

```
# Create a virtual environment (required since PEP 668)
python3 -m venv ~/venv
source ~/venv/bin/activate  # macOS/Linux
# or
~/venv/Scripts/activate     # Windows
```

(continues on next page)

(continued from previous page)

```
# Install dsviper
pip install dsviper
```

Installing from the DevKit

If you also want the CLI tools, templates, and offline documentation, download the DevKit from the Downloads page, unzip, and run:

```
cd dsviper-*-devkit
pip install -r requirements.txt
```

The DevKit's `requirements.txt` pulls `dsviper` from PyPI alongside the GUI/web dependencies needed by the bundled tools.

Verifying Installation

Verify that dsviper is correctly installed. `version()` returns a `(major, minor, patch)` tuple:

```
>>> import dsviper
>>> v = dsviper.version()
>>> isinstance(v, tuple) and len(v) == 3
True
```

Understanding dsviper

dsviper is the Python extension module that provides access to the Viper runtime. Key points:

- **dsviper = Viper**: Learning dsviper means learning Viper. The Python API mirrors the C++ API.
- **Strong typing**: Unlike typical Python, dsviper raises exceptions immediately on type mismatches. This enforces data integrity.
- **Seamless bridge**: Python natives (lists, dicts, tuples) are accepted as input since Viper's metadata drives automatic conversion.

Strong-Typed Layer Over Python

> Unlike Python's permissive duck typing, dsviper enforces types immediately. When you create a `Vector<Int64>`, only integers can be added. Type mismatches raise `ViperError` exceptions at the point of error, not downstream when data is corrupted. See *Error Handling* for details.

Quick Test

Test the type system. Create a vector of strings:

```
>>> from dsviper import *

>>> v = Value.create(TypeVector(Type.STRING))
>>> v.append("hello")
>>> v.append("world")
>>> v
['hello', 'world']
```

Type checking in action — adding the wrong type raises immediately:

```
>>> v.append(42)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...expected type 'str', got 'int'...
```

What's Next

- *Error Handling* - Understanding ViperError exceptions
- *Types and Values* - Understanding the type system
- *DSM* - Define data models with DSM
- *Tutorial* - Complete walkthrough with User/Login example

Tutorial

This tutorial walks through a complete example: creating a data model, setting up a database, and performing operations with CommitDatabase.

> **Prerequisites**: This tutorial assumes you have read *DSM* and understand the basics of DSM syntax and the assemble → parse → introspect workflow.

The User/Login Model

We'll create a simple model with Users who have Login credentials and Identity information.

Step 1: Define the Data Model

Create a file `model.dsm`:

```
namespace Tuto {f529bc42-0618-4f54-a3fb-d55f95c5ad03} {

concept User;

struct Login {
    string nickname;
    string password;
};

struct Identity {
    string firstname;
    string lastname;
};

attachment<User, Login> login;
attachment<User, Identity> identity;

};
```

Step 2: Validate the Model

Check the DSM syntax:

```
python3 tools/dsm_util.py check model.dsm
```

Step 3: Create a Database

Create a Commit database that embeds the definitions:

```
python3 tools/dsm_util.py create_commit_database model.dsm model.cdb
```

Step 4: Open and Explore

Open the database in Python and list the types and attachments it carries:

```
>>> sorted(str(t) for t in db.definitions().types())
['Tuto::Account', 'Tuto::Identity', 'Tuto::Login', 'Tuto::Status', 'Tuto::Texture',
↪'Tuto::Thumbnail', 'Tuto::User']

>>> sorted(str(a).split()[-1] for a in db.definitions().attachments())
['Tuto::account', 'Tuto::avatar', 'Tuto::identity', 'Tuto::login', 'Tuto::portrait']
```

Step 5: Inject Constants

`db.definitions().inject()` makes types accessible as constants in the caller's namespace (already done in this tutorial's setup):

```
>>> TUTO_A_USER_LOGIN
attachment<User, Login> Tuto::login

>>> TUTO_S_LOGIN
Tuto::Login
```

Naming convention: Constants follow the pattern `{NAMESPACE}_{KIND}_{NAME}`:

Kind	Prefix	Example	Description
Attachment	`_A_`	`TUTO_A_USER_LOGIN`	Attachment type
Structure	`_S_`	`TUTO_S_LOGIN`	Structure type
Enumeration	`_E_`	`TUTO_E_STATUS`	Enumeration type
Concept	`_C_`	`TUTO_C_USER`	Concept type
Path	`_P_`	`TUTO_P_LOGIN_NICKNAME`	Path to field

Step 6: Create a Key and Document

Create a new User key. `instance_id()` returns the underlying UUID (randomly generated, so we just check the type here):

```
>>> key = TUTO_A_USER_LOGIN.create_key()
>>> isinstance(key.instance_id(), ValueUUId)
True
```

Create a Login document:

```
>>> login = TUTO_A_USER_LOGIN.create_document()
>>> login
{nickname='', password=''}

>>> login.nickname = "zoop"
>>> login.password = "robust"
>>> login
{nickname='zoop', password='robust'}
```

Step 7: Commit to Database

Create a mutable state, associate the document with the key, and commit:

```
>>> mutable_state = CommitMutableState(db.initial_state())

>>> mutable_state.attachment_mutating().set(TUTO_A_USER_LOGIN, key, login)

>>> commit_id = db.commit_mutations("First Commit", mutable_state)
>>> isinstance(commit_id, ValueCommitId) and len(str(commit_id)) == 40
True
```

Important

The Dual-Layer Contract

CommitDatabase guarantees structural integrity but NOT semantic integrity. When concurrent streams converge, mutations on non-existent documents or unresolved paths are silently ignored. Your application must validate business rules when consuming state.

See *The Dual-Layer Contract*.

Step 8: Read from Database

Read the document back:

```
>>> state = db.state(commit_id)
>>> result = state.attachment_getting().get(TUTO_A_USER_LOGIN, key)
>>> result
Optional({nickname='zoop', password='robust'})

>>> result.unwrap()
{nickname='zoop', password='robust'}
```

Step 9: Update a Field

Update using a path-based setter. Capture the new commit id returned by `commit_mutations` — the database's mutating APIs don't auto-advance an implicit "current commit" pointer, so chaining commits requires the explicit id:

```
>>> mutable_state = CommitMutableState(db.state(commit_id))
>>> mutable_state.attachment_mutating().update(TUTO_A_USER_LOGIN, key, TUTO_P_LOGIN_
```

(continues on next page)

(continued from previous page)

```
↪NICKNAME, "zoopy")
>>> updated_id = db.commit_mutations("Update Nickname", mutable_state)
```

Step 10: View History

Read from different commits:

```
>>> state = db.state(updated_id)
>>> state.attachment_getting().get(TUTO_A_USER_LOGIN, key)
Optional({nickname='zoopy', password='robust'})

>>> state = db.state(commit_id)
>>> state.attachment_getting().get(TUTO_A_USER_LOGIN, key)
Optional({nickname='zoop', password='robust'})

>>> state = db.initial_state()
>>> state.attachment_getting().get(TUTO_A_USER_LOGIN, key)
nil
```

Step 11: Inspect Commit Headers

```
>>> header = db.commit_header(updated_id)
>>> header.label()
'Update Nickname'
>>> header.parent_commit_id() == commit_id
True
```

Two Approaches

Viper supports two ways to work with your data model:

Dynamic API (Above)

Use Viper's runtime metadata directly:

- `DSMBuilder.assemble(path).parse()` → work with types at runtime
- Flexible, interpretive, Python-friendly
- No code generation needed

Static API (Generated)

Generate infrastructure code from your DSM:

```
python3 tools/dsm_util.py create_python_package model.dsm
```

This creates a Python package with:

- Type-safe classes for concepts and structures
- Generated accessors for attachments
- IDE autocompletion support

```
>>> import model.attachments as ma

# Use generated types
>>> key = ma.Tuto_UserKey.create()
>>> login = ma.Tuto_Login()
>>> login.nickname = "user"

# Use generated accessors
>>> state = CommitMutableState(db.state(db.last_commit_id()))
>>> ma.tuto_user_login_set(state.attachment_mutating(), key, login)
```

Note

The Static API snippet above requires running Kibo first to generate the `model.attachments` package. It is illustrative — see the *Kibo* chapter for the full code-generation workflow.

Both approaches use the same underlying Viper runtime.

What's Next

- *Database* - Database and CommitDatabase in detail
- *Blobs* - Working with binary data
- *Types and Values* - Deep dive into the type system

Types and Values

The Viper type system provides strong typing with runtime validation. This chapter covers all available types and how to create values.

The Universal Constructor

Create any value using `Value.create(type, [initial_value])`:

```
>>> from dsviper import *

>>> Value.create(TypeFloat())
0.0

>>> Value.create(Type.STRING, "hello")
'hello'

>>> Value.create(TypeVector(Type.INT64), [1, 2, 3])
[1, 2, 3]
```

If `initial_value` is omitted, the value is initialized to the type's "zero".

Choosing the Right Constructor

Viper offers multiple ways to create values. Use this guide to pick the right one:

Pattern	When to Use	Example
`Type.STRING`	Primitive type constant	`Value.create(Type.STRING, "hello")`
`TypeVector(...)`	Parameterized container	`Value.create(TypeVector(Type.INT64), [1,2,3])`
`ValueString("...")`	Direct value (known type)	`ValueString("hello")`
`Value.create(t, v)`	Universal factory	`Value.create(t_struct, {"field": 1})`
`Value.deduce(v)`	Let Viper infer type	`Value.deduce([1, 2, 3])`
`ValueUUId.create()`	Factory with generation	`ValueUUId.create()`

Quick rules:

- Use **Type.X** for primitives: `Type.STRING`, `Type.INT64`, `Type.BOOL`
- Use **TypeX(…)** for containers: `TypeVector(...)`, `TypeMap(...)`, `TypeOptional(...)`
- Use **ValueX(…)** when type is known and you want direct construction
- Use **Value.create(t, v)** when working with dynamic types or DSM definitions
- Use **Value.deduce(v)** for quick prototyping (type inferred from Python value)

Type Constants

Common types are available as constants on `Type`:

Constant	Type
`Type.BOOL`	Boolean
`Type.INT8` to `Type.INT64`	Signed integers
`Type.UINT8` to `Type.UINT64`	Unsigned integers
`Type.FLOAT`	32-bit float
`Type.DOUBLE`	64-bit double
`Type.STRING`	UTF-8 string
`Type.UUID`	UUID
`Type.ANY`	Any type

Primitive Types

Boolean

Note that Viper values render with their own `repr` — booleans appear as `true`/`false`, not Python's `True`/`False`.

```
>>> Value.create(Type.BOOL)
false

>>> Value.create(Type.BOOL, True)
true
```

Type checking is enforced:

```
>>> Value.create(Type.BOOL, 4)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...expected type 'bool', got 'int'...
```

Integers

Signed and unsigned integers with range validation:

```
>>> Value.create(Type.INT8)
0

>>> Value.create(Type.INT8, 42)
42

>>> Value.create(Type.INT8, 256)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...value is not in the range of 'int8'...
```

Available types: `INT8`, `INT16`, `INT32`, `INT64`, `UINT8`, `UINT16`, `UINT32`, `UINT64`

Floats

```
>>> Value.create(Type.FLOAT)
0.0

>>> Value.create(Type.FLOAT, 42)
42.0

>>> Value.create(Type.FLOAT, 1e300)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...value is not in the range of 'float'...
```

String

UTF-8 encoded strings:

```
>>> Value.create(Type.STRING)
''

>>> Value.create(Type.STRING, "hello world")
'hello world'
```

UUID

Generate a fresh UUID — `ValueUUId.create()` returns a randomly-generated value:

```
>>> uuid = ValueUUId.create()
>>> isinstance(str(uuid), str) and str(uuid).count('-') == 4
True
```

Parse a known UUID string:

```
>>> uuid = ValueUUId.create("c98ddeec-0496-494c-b1e9-b470ff204be3")
>>> uuid
c98ddeec-0496-494c-b1e9-b470ff204be3
```

Malformed UUIDs are rejected:

```
>>> ValueUUId.create("invalid")
Traceback (most recent call last):
    ...
dsviper.ViperError: ...malformed UUId...
```

Mathematical Types

Vec

Fixed-size numeric arrays:

```
>>> t = TypeVec(Type.FLOAT, 3)
>>> Value.create(t)
(0.0, 0.0, 0.0)

>>> Value.create(t, (1, 2, 3))
(1.0, 2.0, 3.0)
```

Mat

Matrices with columns and rows:

```
>>> t = TypeMat(Type.FLOAT, 2, 3)
>>> Value.create(t)
[(1.0, 0.0, 0.0), (0.0, 1.0, 0.0)]

>>> Value.create(t, ((1, 2, 3), (4, 5, 6)))
[(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)]
```

Type Deduction

Let Python infer the Viper type with `Value.deduce()`:

```
>>> v = Value.deduce([1, 2, 3])
>>> v.type()
vector<int64>

>>> v = Value.deduce([1, 2, (3, "string"), {1.0, 2.0}])
>>> v.type()
vector<int64|set<double>|tuple<int64, string>>
```

Type Information

Every value knows its type:

```
>>> v = Value.create(TypeVector(Type.STRING), ["a", "b"])
>>> v.type()
vector<string>

>>> v.description()
"['a', 'b']:vector<string>"
```

Type Checking

Viper validates types at runtime:

```
>>> v = Value.create(TypeVector(Type.STRING))
>>> v.append("valid")
>>> v.append(123)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...expected type 'str', got 'int'...
```

The Seamless Bridge

Python natives are automatically converted:

```
>>> v = Value.create(TypeVector(Type.STRING))
>>> v.extend(["from", "python", "list"])
>>> v
['from', 'python', 'list']

>>> m = Value.create(TypeMap(Type.STRING, Type.INT64), {"a": 1, "b": 2})
>>> m
{'a': 1, 'b': 2}
```

The bridge uses type metadata to validate and convert Python objects.

What's Next

- *Collections* - Vectors, Maps, Sets, and more
- *Structures and Enumerations* - User-defined types

Collections

Viper provides strongly-typed collections that mirror Python's built-in collections while enforcing type safety.

Vector

`ValueVector` is compatible with Python's list API:

```
>>> from dsviper import *

>>> v = Value.create(TypeVector(Type.STRING), ["hello", "world"])
```

(continues on next page)

(continued from previous page)

```
>>> v
['hello', 'world']
```

Append a single element:

```
>>> v.append("!")
>>> v
['hello', 'world', '!']
```

Extend with a Python list:

```
>>> v.extend(["from", "python"])
>>> v
['hello', 'world', '!', 'from', 'python']
```

Index access (positive and negative):

```
>>> v[0]
'hello'
>>> v[-1]
'python'
```

Length:

```
>>> len(v)
5
```

Iterate:

```
>>> for item in v:
...     print(item)
hello
world
!
from
python
```

Note

`ValueVector` does not currently support slice access (`v[1:3]`). Iterate explicitly or use `list(v)` to materialize a slice in Python.

Type Safety

```
>>> v.append(42)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...expected type 'str', got 'int'...
```

Set

`ValueSet` is compatible with Python's set API:

```
>>> s = Value.create(TypeSet(Type.INT64), {1, 2, 3, 2, 1})
>>> s
{1, 2, 3}
```

Add and remove:

```
>>> s.add(4)
>>> s.remove(1)
>>> s
{2, 3, 4}
```

Membership and length:

```
>>> 2 in s
True
>>> len(s)
3
```

Set operations:

```
>>> s2 = Value.create(TypeSet(Type.INT64), {3, 4, 5})
>>> s.union(s2)
{2, 3, 4, 5}
>>> s.intersection(s2)
{3, 4}
```

Map

`ValueMap` is compatible with Python's dict API:

```
>>> m = Value.create(TypeMap(Type.STRING, Type.INT64), {"a": 1, "b": 2})
>>> m
{'a': 1, 'b': 2}
```

Get and set:

```
>>> m["c"] = 3
>>> m["a"]
1
```

Keys, values, items:

```
>>> list(m.keys())
['a', 'b', 'c']
>>> list(m.values())
[1, 2, 3]
>>> list(m.items())
[('a', 1), ('b', 2), ('c', 3)]
```

Delete and length:

```
>>> del m["a"]
>>> len(m)
2
```

Complex Keys

Maps can use complex types as keys:

```
>>> t = TypeMap(TypeTuple([Type.INT64, Type.INT64]), Type.STRING)
>>> m = Value.create(t, {(1, 2): "one-two", (3, 4): "three-four"})
>>> m[(1, 2)]
'one-two'
```

Optional

`ValueOptional` holds a value or nothing:

```
>>> opt = Value.create(TypeOptional(Type.STRING))
>>> opt
nil

>>> opt.is_nil()
True
```

Wrap a value:

```
>>> opt.wrap("hello")
>>> opt
Optional('hello')

>>> opt.is_nil()
False
```

Unwrap:

```
>>> opt.unwrap()
'hello'
```

Unwrapping an empty optional raises an error:

```
>>> empty = Value.create(TypeOptional(Type.STRING))
>>> empty.unwrap()
Traceback (most recent call last):
    ...
dsviper.ViperError: ...Try to unwrap empty optional<string>...
```

Initialize with Value

```
>>> opt = Value.create(TypeOptional(Type.INT64), 42)
>>> opt
Optional(42)
```

Tuple

`ValueTuple` holds heterogeneous values. Note that booleans render with Viper's own repr (`true`/`false`), not Python's `True`/`False`:

```
>>> t = TypeTuple([Type.STRING, Type.INT64, Type.BOOL])
>>> v = Value.create(t, ("hello", 42, True))
>>> v
('hello', 42, true)
```

Access by index:

```
>>> v[0]
'hello'
>>> v[1]
42
```

Variant

`ValueVariant` holds one value from a set of possible types:

```
>>> t = TypeVariant([Type.STRING, Type.INT64])
>>> v = Value.create(t, "a string")
>>> v.type()
string|int64
```

Change the value:

```
>>> v.wrap(42)
>>> v.unwrap()
42
```

Wrong type is rejected:

```
>>> v.wrap(3.14)
Traceback (most recent call last):
    ...
dsviper.ViperError: ...expected type 'string|int64', got 'float'...
```

XArray

Vector vs XArray: When to Use Each

Feature	Vector	XArray
Indexing	Integer indices (0, 1, 2…)	UUID positions
Insert/Remove	Indices shift	Positions stable
Multiplayer editing	Last-write-wins	Merge-friendly
Performance	Faster for local use	Slight overhead
Use case	Local arrays, batch processing	Shared editable lists

Why XArray? In multiplayer scenarios, two users might insert at "index 3" simultaneously. With Vector, one insert wins and the other is lost or corrupted. With XArray, each element has a UUID position that remains stable across merges.

Example: A shared todo list where multiple users add/remove items should use XArray. A local computation buffer should use Vector.

XArray Basics

`ValueXArray` preserves order during concurrent mutations using UUID positions instead of integer indices.

Create and append. Note that `append()` returns `x.END` — the zero-UUID sentinel that means "the end of the array" — not the position of the just-appended element:

```
>>> t = TypeXArray(Type.STRING)
>>> x = Value.create(t)

>>> x.END
00000000-0000-0000-0000-000000000000

>>> x.append("first")
00000000-0000-0000-0000-000000000000
>>> x.append("second")
00000000-0000-0000-0000-000000000000
>>> x
['first', 'second']
```

Recover the actual UUID position of an element with `position(index)` or `position_of(value)`, then access it with `at(pos)` (or `x[pos]`):

```
>>> pos0 = x.position(0)
>>> x.at(pos0)
'first'
>>> x[pos0]
'first'

>>> p = x.position_of("second")
>>> x.at(p)
'second'
```

`insert(pos, value)` inserts **before** the given position and returns the UUID position of the new element. Inserting at `x.END` is therefore equivalent to appending; inserting at `pos0` prepends:

```
>>> p_third = x.insert(x.END, "third")
>>> x.at(p_third)
'third'

>>> p_zero = x.insert(pos0, "zero")
>>> x.at(p_zero)
'zero'

>>> x
['zero', 'first', 'second', 'third']
```

Iterate over values, or over `(position, value)` pairs:

```
>>> for item in x:
...     print(item)
```

(continues on next page)

(continued from previous page)

```
zero
first
second
third
```

Any

`TypeAny` accepts any value:

```
>>> v = Value.create(TypeVector(Type.ANY))
>>> v.append("a string")
>>> v.append(42)
>>> v.append([1, 2, 3])

>>> v.description()
"['a string':string:any, 42:int64:any, [1, 2, 3]:vector<int64>:any]:vector<any>"
```

Nested Collections

Collections can be nested:

```
>>> t = TypeVector(TypeMap(Type.STRING, Type.INT64))
>>> v = Value.create(t)
>>> v.append({"a": 1})
>>> v.append({"b": 2, "c": 3})
>>> v
[{'a': 1}, {'b': 2, 'c': 3}]
```

What's Next

- *Structures and Enumerations* - User-defined types
- *Database* - Persisting collections

Structures and Enumerations

Structures and enumerations are user-defined types that require registration with `Definitions` before use. This chapter covers creating and working with both.

Creating Structures with Definitions

Structures are defined within a `Definitions` context:

```
>>> from dsviper import *

>>> defs = Definitions()
>>> ns = NameSpace(ValueUUId("f529bc42-0618-4f54-a3fb-d55f95c5ad03"), "Tuto")
```

Define a structure with a descriptor:

```
>>> desc = TypeStructureDescriptor("Login")
>>> desc.add_field("nickname", Type.STRING)
```

(continues on next page)

(continued from previous page)

```
>>> desc.add_field("password", Type.STRING)

>>> t_login = defs.create_structure(ns, desc)
>>> t_login
Tuto::Login
```

Instantiating Structures

Create structure instances with `Value.create()`. Default values use the type's "zero":

```
>>> login = Value.create(t_login)
>>> login
{nickname='', password=''}
```

Initialize from a dict:

```
>>> login = Value.create(t_login, {"nickname": "zoop"})
>>> login
{nickname='zoop', password=''}
```

Field Access

Access fields by name:

```
>>> login.nickname
'zoop'

>>> login.password = "secret"
>>> login
{nickname='zoop', password='secret'}
```

Type Checking

Field assignments are type-checked:

```
>>> login.nickname = 42
Traceback (most recent call last):
    ...
dsviper.ViperError: ...expected type 'str', got 'int'...
```

Default Values

Specify default values when defining fields:

```
>>> desc = TypeStructureDescriptor("Config")
>>> desc.add_field("scale", ValueFloat(1.0))
>>> desc.add_field("items", Value.create(TypeVector(Type.INT64), [1, 2, 3]))

>>> t_config = defs.create_structure(ns, desc)
>>> Value.create(t_config)
{scale=1.0, items=[1, 2, 3]}
```

Nested Structures

Structures can contain other structures:

```
>>> desc_pos = TypeStructureDescriptor("Position")
>>> desc_pos.add_field("x", Type.FLOAT)
>>> desc_pos.add_field("y", Type.FLOAT)
>>> t_position = defs.create_structure(ns, desc_pos)

>>> desc_vertex = TypeStructureDescriptor("Vertex")
>>> desc_vertex.add_field("position", t_position)
>>> desc_vertex.add_field("label", Type.STRING)
>>> t_vertex = defs.create_structure(ns, desc_vertex)

>>> vertex = Value.create(t_vertex)
>>> vertex
{position={x=0.0, y=0.0}, label=''}
```

Mutate a nested field:

```
>>> vertex.position.x = 10.0
>>> vertex.position.y = 20.0
>>> vertex.label = "A"
>>> vertex
{position={x=10.0, y=20.0}, label='A'}
```

Paths

A `Path` locates a piece of information within a value:

```
>>> p = Path.from_field("nickname").const()
>>> p
.nickname
```

Apply a path to read or write the targeted field:

```
>>> p.at(login)
'zoop'

>>> p.set(login, "new_nick")
>>> login.nickname
'new_nick'
```

Complex Paths

Paths can traverse nested structures:

```
>>> p = Path.from_field("position").field("x").const()
>>> p
.position.x

>>> p.at(vertex)
10.0
>>> p.set(vertex, 15.0)
```

Path Components

Paths can include:

- Field access: `.field("name")`
- Index access: `.index(0)`
- Unwrap: `.unwrap()`
- Map key: `.key(key_value)`

```
>>> p = Path.from_field("items").index(0).const()
>>> p
.items[0]

>>> p.components()
[{type: Field, value: 'items':string}, {type: Index, value: 0:uint64}]
```

Enumerations

Enumerations are types with a fixed set of named cases.

Defining an Enumeration

```
>>> desc = TypeEnumerationDescriptor("Status", documentation="Task status")
>>> desc.add_case("pending", "Waiting to start")
>>> desc.add_case("active", "In progress")
>>> desc.add_case("completed", "Finished")

>>> t_status = defs.create_enumeration(ns, desc)
>>> t_status
Tuto::Status
```

Creating Enumeration Values

By case name (most common):

```
>>> status = ValueEnumeration(t_status, "active")
>>> status.name()
'active'
```

By index (0-based):

```
>>> status = ValueEnumeration(t_status, 0)
>>> status.name()
'pending'
```

Default (first case):

```
>>> Value.create(t_status)
.pending
```

Specify case with `Value.create()`:

```
>>> Value.create(t_status, "completed")
.completed
```

Properties

```
>>> status = ValueEnumeration(t_status, "active")

>>> status.name()
'active'

>>> status.index()
1

>>> status.type_enumeration()
Tuto::Status
```

Type Introspection

Query available cases from the type:

```
>>> cases = t_status.cases()
>>> [c.name() for c in cases]
['pending', 'active', 'completed']
```

Query a case by name:

```
>>> case = t_status.query("active")
>>> case.documentation()
'In progress'
```

Check existence (raises on invalid):

```
>>> t_status.check("unknown")
Traceback (most recent call last):
    ...
dsviper.ViperError: ...the case unknown is not defined for enum Tuto::Status...
```

Comparison

Enumerations compare lexicographically by name, not by index:

```
>>> pending = ValueEnumeration(t_status, "pending")
>>> active = ValueEnumeration(t_status, "active")

>>> active < pending
True
```

Constraints

- Maximum 256 cases per enumeration (uint8 index)
- Case names must be unique within an enumeration

- Empty enumerations are not allowed

Type Information

Introspect structure types. Field reprs include the field type:

```
>>> t_login.fields()
[nickname:string, password:string]

>>> field = t_login.fields()[0]
>>> field.name()
'nickname'
>>> field.type()
string
```

Using Structures with Databases

Structures are typically stored via attachments. See *DSM* for defining attachments and *Database* for persistence patterns.

Note

The "Quick preview" snippet below depends on a DSM model compiled by Kibo (`TUTO_A_USER_LOGIN` is generated). It is illustrative — see the *Tutorial* for a self-contained working example.

```
# With DSM definitions loaded
>>> key = TUTO_A_USER_LOGIN.create_key()
>>> login = TUTO_A_USER_LOGIN.create_document()
>>> login.nickname = "alice"

# Store in database
>>> db.set(TUTO_A_USER_LOGIN, key, login)
```

What's Next

- *DSM* - Define data models with attachments
- *Database* - Persisting structures
- *Serialization* - JSON and binary encoding

Error Handling

Viper uses a single exception type for all errors: `ViperError`. This chapter explains how to catch, interpret, and recover from errors in your Python code.

The ViperError Exception

All Viper operations that fail raise a `ViperError` exception:

```
from dsviper import ViperError, ValueInt8

try:
```

(continues on next page)

(continued from previous page)

```
    v = ValueInt8(200)  # Out of range for int8
except ViperError as e:
    print(str(e))
```

Output:

```
[macbook@myapp]:Viper:Value:1:value is not in the range of 'int8'
```

The message follows the format: `[host@process]:Component:Domain:Code:Message`

Parsing Error Details

To access individual error fields, use the `Error` class to parse the exception message:

```
from dsviper import ViperError, Error, ValueInt8

try:
    v = ValueInt8(200)
except ViperError as e:
    error = Error.parse(str(e))
    if error:
        print(error.component())  # "Viper"
        print(error.domain())  # "Value"
        print(error.code())  # 1
        print(error.message())  # "value is not in the range of 'int8'"
        print(error.hostname())  # "macbook"
        print(error.process_name())  # "myapp"
```

Error Class Methods

Method	Return	Description
`Error.parse(str)`	`Error` or `None`	Parse an error string into an Error object
`error.component()`	`str`	The subsystem (e.g., “Viper”, “Database”)
`error.domain()`	`str`	The functional domain (e.g., “Value”, “Type”)
`error.code()`	`int`	Numeric error code within the domain
`error.message()`	`str`	Human-readable description
`error.hostname()`	`str`	Host where error originated
`error.process_name()`	`str`	Process name
`error.explained()`	`str`	Full formatted error string

Common Error Categories

Type Mismatch Errors

These occur when you provide a value of the wrong type:

```
from dsviper import Value, Type, TypeVector, ViperError

```

(continues on next page)

(continued from previous page)

```
t_vec = TypeVector(Type.INT64)

try:
    # Trying to create a vector with strings instead of integers
    v = Value.create(t_vec, ["a", "b", "c"])
except ViperError as e:
    print(str(e))  # "...expected type 'long', got 'str'..."
```

Recovery: Check your input types before calling Viper functions.

Range Validation Errors

These occur when numeric values exceed their type's range:

```
from dsviper import ValueInt8, ViperError

# Int8 range: -128 to 127
try:
    v = ValueInt8(200)
except ViperError as e:
    print(str(e))  # "...value is not in the range of 'int8'..."
```

Integer type ranges:

Type	Range
`Int8`	-128 to 127
`Int16`	-32,768 to 32,767
`Int32`	-2^{31} to $2^{31}-1$
`Int64`	-2^{63} to $2^{63}-1$
`UInt8`	0 to 255
`UInt16`	0 to 65,535
`UInt32`	0 to $2^{32}-1$
`UInt64`	0 to $2^{64}-1$

Recovery: Validate numeric inputs before creating values, or use larger types.

Optional Unwrap Errors

These occur when you try to unwrap an empty (nil) optional:

```
from dsviper import Value, Type, TypeOptional, ValueOptional

t_opt = TypeOptional(Type.STRING)
v_empty = ValueOptional(t_opt)  # Creates nil optional

try:
    content = v_empty.unwrap()
except Exception as e:
    print("Cannot unwrap nil optional")
```

Recovery: Always check `is_nil()` before calling `unwrap()`:

```
if not v_empty.is_nil():
    content = v_empty.unwrap()
else:
    content = "default value"
```

Cast Errors

These occur when you cast a value to an incompatible type:

```
from dsviper import ValueString, ValueInt64, ViperError

v_str = ValueString("hello")

try:
    v_int = ValueInt64.cast(v_str)
except ViperError as e:
    print(str(e))  # "...expected int64, got string [cast]..."
```

Recovery: Use `type()` to check the value's type before casting:

```
from dsviper import Type

if v_str.type() == Type.INT64:
    v_int = ValueInt64.cast(v_str)
```

Key Not Found (Not an Error)

When accessing non-existent keys in databases, Viper does **not** raise an exception. Instead, `get()` returns a `ValueOptional` that is nil:

```
# db.get() returns ValueOptional, not the value directly
result = db.get(attachment, unknown_key)

# Check if key exists
if result.is_nil():
    print("Key not found")
else:
    value = result.unwrap()
```

> **Note**: `ValueOptional` is a Viper container type, unrelated to Python's `typing.Optional`. It wraps a value with nil/non-nil semantics and provides `is_nil()`, `unwrap()`, and `wrap()` methods.

Alternatives: Use `has()` to check key existence before get:

```
if db.has(attachment, key):
    value = db.get(attachment, key).unwrap()
```

DSM Parse Errors

These occur when parsing invalid DSM files:

```
from dsviper import DSMBuilder
```

(continues on next page)

(continued from previous page)

```
builder = DSMBuilder.assemble("invalid.dsm")
report, dsm_defs, defs = builder.parse()

if report.has_error():
    for error in report.errors():
        print(f"Line {error.line()}: {error.message()}")
```

Note: DSM parsing returns a report object instead of raising exceptions, allowing you to collect multiple errors at once.

Best Practices

1. Catch Specific Errors

Always catch `ViperError` specifically, not bare `Exception`:

```
# Good
try:
    v = ValueInt8(value)
except ViperError as e:
    handle_viper_error(e)

# Bad - catches too much
try:
    v = ValueInt8(value)
except Exception:
    pass
```

2. Check Before You Leap

For performance-critical code, validate inputs before calling Viper:

```
# Check range before creating value
def safe_int8(value: int) -> ValueInt8:
    if not (-128 <= value <= 127):
        raise ValueError(f"Value {value} out of int8 range")
    return ValueInt8(value)
```

3. Use Optionals Correctly

Never assume an optional contains a value:

```
# Good - check before unwrap
result = db.get(attachment, key)
if not result.is_nil():
    value = result.unwrap()
else:
    value = default

# Bad - will raise if nil
value = db.get(attachment, key).unwrap()
```

4. Handle DSM Errors Gracefully

Always check the parse report before using definitions:

```
builder = DSMBuilder.assemble(dsm_path)
report, dsm_defs, defs = builder.parse()

if report.has_error():
    for err in report.errors():
        print(f"Error at line {err.line()}: {err.message()}")
    raise RuntimeError("DSM parsing failed")

# Safe to use defs here
```

5. Transaction Error Handling

Wrap database transactions with proper error handling:

```
db.begin_transaction()
try:
    db.set(attachment, key, value)
    db.commit()
except ViperError as e:
    db.rollback()
    raise
```

Debugging Tips

Enable Verbose Logging

Use Viper's logging system to trace operations:

```
from dsviper import LoggerConsole, Logging

# Create a console logger with debug level
logger = LoggerConsole(Logging.LEVEL_DEBUG)
logging = logger.logging()

# Log operations
logging.info("Starting database operation")
logging.error("Operation failed")
```

Available log levels: `LEVEL_ALL`, `LEVEL_DEBUG`, `LEVEL_INFO`, `LEVEL_WARNING`, `LEVEL_ERROR`, `LEVEL_CRITICAL`.

Inspect Error Fields

When debugging, parse and print all error fields:

```
except ViperError as e:
    error = Error.parse(str(e))
    if error:
        print(f"Host: {error.hostname()}")
```

(continues on next page)

(continued from previous page)

```
        print(f"Process: {error.process_name()}")
        print(f"Component: {error.component()}")
        print(f"Domain: {error.domain()}")
        print(f"Code: {error.code()}")
        print(f"Message: {error.message()}")
```

Remote Errors

When using `DatabaseRemote` or `ServiceRemote`, errors include the remote host information, helping you identify where the error occurred:

```
[remote-server@db-process]:Viper:Database:3:Connection timeout
```

What's Next

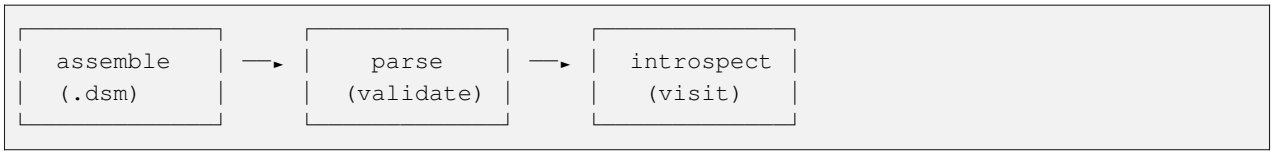
- *Types and Values* - Learn the type system
- *Collections* - Work with containers

DSM Processing

This chapter covers loading and processing DSM (Digital Substrate Model) files in Python.

The DSM Workflow

Processing DSM files follows three steps:



Step 1: Assemble

`DSMBuilder.assemble()` reads DSM files. The path can be a single `.dsm` file or a directory containing `.dsm` files; in the directory case, every file is concatenated and parsed as a single unit. The doctests below reuse the bundled `Tuto` fixture builder pre-loaded by the test harness:

```
>>> builder = _builder
>>> [p.source().split('/')[-1] for p in builder.parts()]
['model.dsm']
```

In production code you would assemble from your own paths:

```
>>> builder = DSMBuilder.assemble("model.dsm")

>>> builder = DSMBuilder.assemble("models")

>>> for part in builder.parts():
...     print(part.source())
```

Step 2: Parse

`parse()` validates syntax and semantics, returning three values:

```
>>> report, dsm_defs, defs = builder.parse()
>>> report.has_error()
False
>>> type(dsm_defs).__name__
'DSMDefinitions'
>>> type(defs).__name__
'DefinitionsConst'
```

Return Value	Type	Description
`report`	`DSMParseReport`	Errors and warnings
`dsm_defs`	`DSMDefinitions`	Structured DSM data (or None if errors)
`defs`	`DefinitionsConst`	Runtime definitions (or None if errors)

Handling Errors

```
>>> report, dsm_defs, defs = builder.parse()
>>> if report.has_error():
...     for error in report.errors():
...         print(f"{error.source()}:{error.line()}: {error.message()}")
... else:
...     print("Parse successful!")
```

Step 3: DSM Introspection

`DSMDefinitions` provides structured access to inspect the parsed model.

DSMDefinitions

The root container for all DSM elements:

```
>>> dsm_defs.concepts()
[Tuto::User]

>>> sorted(str(s) for s in dsm_defs.structures())
['Tuto::Account', 'Tuto::Identity', 'Tuto::Login', 'Tuto::Texture', 'Tuto::Thumbnail']

>>> dsm_defs.enumerations()
[Tuto::Status]

>>> sorted(str(a).split()[-1] for a in dsm_defs.attachments())
['Tuto::account', 'Tuto::avatar', 'Tuto::identity', 'Tuto::login', 'Tuto::portrait']
```

DSMConcept

Inspect concept definitions:

```
>>> concept = dsm_defs.concepts()[0]
>>> concept.type_name()
Tuto::User

>>> isinstance(concept.runtime_id(), ValueUUId)
True

>>> concept.documentation()
'A user.'

>>> concept.parent() is None
True
```

For a model with concept inheritance, `parent()` returns the parent concept, e.g. `Tuto::Admin`.

DSMStructure

Inspect structure definitions and their fields:

```
>>> struct = [s for s in dsm_defs.structures() if str(s.type_name()) == 'Tuto::Login
↪'][0]
>>> struct.type_name()
Tuto::Login

>>> [(f.name(), str(f.type())) for f in struct.fields()]
[('nickname', 'string'), ('password', 'string')]

>>> struct.fields()[0].documentation()
''
```

DSMEnumeration

Inspect enumeration definitions and their cases:

```
>>> enum = dsm_defs.enumerations()[0]
>>> enum.type_name()
Tuto::Status

>>> [case.name() for case in enum.members()]
['pending', 'active', 'completed']
```

The Tuto fixture ties `Status` into a struct used by the `account` attachment, so the enum can be exercised end-to-end through the runtime:

```
>>> account = TUTO_A_USER_ACCOUNT.create_document()
>>> account
{state=.pending}

>>> account.state = ValueEnumeration(TUTO_E_STATUS, "active")
>>> account
{state=.active}
```

DSMAttachment

Inspect attachment definitions. `identifier()` returns the fully qualified name `<concept>.<attachment>`:

```
>>> att = dsm_defs.attachments()[0]
>>> att.identifier()
'Tuto::User.login'

>>> att.key_type()
Tuto::User

>>> att.document_type()
Tuto::Login
```

Generate DSM Text

Reconstruct DSM source from definitions. The output groups types by attachment and adds explanatory comments:

```
>>> source = dsm_defs.to_dsm()
>>> 'namespace Tuto' in source and 'concept User' in source
True
>>> 'attachment<User, Login> login' in source
True
```

Pass `show_runtime_id=True` to append runtime IDs.

Using Runtime Definitions

The `DefinitionsConst` from parse enables runtime operations.

Inject Constants

`defs.inject()` makes generated constants available in the namespace. The `Tuto` constants are already in scope thanks to the doctest fixture:

```
>>> TUTO_S_LOGIN
Tuto::Login

>>> TUTO_A_USER_LOGIN
attachment<User, Login> Tuto::login
```

Naming convention: Constants follow the pattern `{NAMESPACE}_{KIND}_{NAME}`:

Kind	Prefix	Example	Description
Attachment	`_A_`	`TUTO_A_USER_LOGIN`	Attachment type
Structure	`_S_`	`TUTO_S_LOGIN`	Structure type
Enumeration	`_E_`	`TUTO_E_STATUS`	Enumeration type
Concept	`_C_`	`TUTO_C_USER`	Concept type
Path	`_P_`	`TUTO_P_LOGIN_NICKNAME`	Path to field

Query Types

```
>>> types = defs.query_types("Login")
>>> types
[Tuto::Login]
>>> Value.create(types[0])
{nickname='', password=''}
```

Access Attachments

```
>>> for att in defs.attachments():
...     print(att.description())
```

Serialization

DSMDefinitions can be serialized for distribution.

Binary (DSMB)

```
>>> blob = dsm_defs.encode()
>>> blob
blob(...)

>>> restored = DSMDefinitions.decode(blob)
>>> type(restored).__name__
'DSMDefinitions'
```

JSON

```
>>> json_str = dsm_defs.json_encode()
>>> 'concepts' in json_str and 'attachments' in json_str
True

>>> restored = DSMDefinitions.json_decode(json_str)
>>> type(restored).__name__
'DSMDefinitions'
```

What's Next

- *Tutorial* - Complete workflow using DSM with CommitDatabase
- *Database* - Persistence with Database and CommitDatabase
- *Serialization* - Binary and JSON encoding

Database

Viper provides two database types for persistence:

Type	Use Case	History
`Database`	Simple key-value storage	No
`CommitDatabase`	Versioned data	Yes (DAG)

This page covers the simple `Database`. For versioned storage, see *CommitDatabase*.

Creating a Database

In-memory and on-disk variants:

```
>>> db = Database.create_in_memory()
>>> db.in_memory()
True
```

```
>>> db = Database.create("data.vdb")

>>> db = Database.create("data.vdb", documentation="Cache for processed meshes")

>>> db = Database.open("data.vdb")

>>> db = Database.open("data.vdb", readonly=True)
```

Before opening, you can check if a file is a valid Database:

```
>>> Database.is_compatible("data.vdb")
True
```

Remote Access

Database supports network access through a server:

```
>>> Database.databases("server.local")
['cache', 'index']

>>> db = Database.connect("cache", "server.local")

>>> db = Database.connect_local("cache", "/tmp/project.sock")
```

See *Server* for deployment details.

Setting Up Definitions

```
>>> _ = db.extend_definitions(_tuto_defs)
>>> sorted(str(a).split()[-1] for a in db.definitions().attachments())
['Tuto::account', 'Tuto::avatar', 'Tuto::identity', 'Tuto::login', 'Tuto::portrait']
```

In your own code you would assemble definitions from your DSM model:

```
>>> builder = DSMBuilder.assemble("model.dsm")
>>> report, dsm_defs, defs = builder.parse()
>>> db.extend_definitions(defs)
>>> defs.inject()
```

CRUD Operations

Create a key and a document:

```
>>> key = TUTO_A_USER_LOGIN.create_key()
>>> login = TUTO_A_USER_LOGIN.create_document()
>>> login.nickname = "alice"
```

Write — all mutations require a transaction:

```
>>> db.begin_transaction()
>>> _ = db.set(TUTO_A_USER_LOGIN, key, login)
>>> db.commit()
```

Read:

```
>>> result = db.get(TUTO_A_USER_LOGIN, key)
>>> result.unwrap()
{nickname='alice', password=''}
```

Check existence:

```
>>> db.has(TUTO_A_USER_LOGIN, key)
True
```

Update:

```
>>> db.begin_transaction()
>>> login.password = "secret"
>>> _ = db.set(TUTO_A_USER_LOGIN, key, login)
>>> db.commit()

>>> db.get(TUTO_A_USER_LOGIN, key).unwrap()
{nickname='alice', password='secret'}
```

List keys for an attachment — `keys()` returns a `ValueSet` of `ValueUUId`:

```
>>> isinstance(db.keys(TUTO_A_USER_LOGIN), ValueSet)
True
>>> len(db.keys(TUTO_A_USER_LOGIN))
1
```

Delete:

```
>>> db.begin_transaction()
>>> _ = db.delete(TUTO_A_USER_LOGIN, key)
>>> db.commit()

>>> db.has(TUTO_A_USER_LOGIN, key)
False
```

Transactions

All write operations require a transaction:

```
>>> db.begin_transaction()
>>> db.in_transaction()
True
>>> db.commit()
>>> db.in_transaction()
False
```

A transaction can also be rolled back:

```
>>> db.begin_transaction()
>>> _ = db.set(TUTO_A_USER_LOGIN, key, login)
>>> db.rollback()
>>> db.has(TUTO_A_USER_LOGIN, key)
False
```

Safe transaction pattern — use try/finally to ensure cleanup:

```
db.begin_transaction()
try:
    db.set(attachment, key, document)
    db.commit()
except ViperError:
    db.rollback()
    raise
```

Lifecycle

Always close a database when done, especially for on-disk databases:

```
>>> tmp_db = Database.create_in_memory()
>>> tmp_db.close()
>>> tmp_db.is_closed()
True
```

Metadata

`Database` exposes a few metadata accessors:

```
>>> db.in_memory()
True
>>> db.path()
'InMemory'
>>> isinstance(db.uuid(), ValueUUId)
True
>>> db.codec_name()
'StreamBinary'
```

For an on-disk database, `path()` returns the file path and `documentation()` returns the string passed to `Database.create(..., documentation=...)`.

Choosing Between Database and CommitDatabase

Feature	Database	CommitDatabase
Simple CRUD	✓	✓
History	✗	✓
Sync	✗	✓
Embedded definitions	✓	✓
Remote access	✓	✓
Blob storage	✓	✓

What's Next

- *CommitDatabase* - Versioned database with history
- *Blobs* - Binary data storage

Commit System

The commit system provides transactional persistence with history tracking.

When to use: Use `CommitDatabase` for versioned persistence with history, concurrent streams, and sync. Every change creates a commit, enabling undo/redo and concurrent editing.

CommitDatabase

`CommitDatabase` provides versioned data storage:

- History is preserved as a DAG of commits
- You can read from any point in history

Important

The Dual-Layer Contract

CommitDatabase guarantees **structural integrity** (deterministic merge, DAG consistency) but NOT **semantic integrity**. Best-effort merge means the state returned after convergence is **structurally sound but semantically untrusted**: mutations may have been silently dropped, and disjoint updates may have combined into a state that violates a cross-field invariant.

Treat post-merge state like deserialized external input: re-validate it at read time, before acting on it. `dsviper.Error` covers only API misuse — it does not signal lost mutations or business-rule violations.

See *The Dual-Layer Contract* for details.

Opening a CommitDatabase

```
>>> db = CommitDatabase.open("model.cdb")
```

To create a new database with embedded definitions, use:

```
python3 tools/dsm_util.py create_commit_database model.dsm model.cdb
```

Reading State

A freshly created database has no commits — `first_commit_id()` and `last_commit_id()` return `None`:

```
>>> db.first_commit_id() is None
True
>>> db.last_commit_id() is None
True
>>> db.head_commit_ids()
set()
```

The `initial_state()` method always works and returns the empty state:

```
>>> initial = db.initial_state()
>>> len(initial.attachment_getting().keys(TUTO_A_USER_LOGIN))
0
```

AttachmentGetting Interface

Read attachments via `attachment_getting()`:

```
>>> getting = state.attachment_getting()

>>> doc = getting.get(attachment, key)
>>> doc
Optional({...})

>>> keys = getting.keys(attachment)
```

Mutations

Create a mutable state and apply changes:

```
>>> mutable_state = CommitMutableState(db.state(db.last_commit_id()))
>>> mutating = mutable_state.attachment_mutating()

>>> mutating.set(attachment, key, document)

>>> mutating.update(attachment, key, path, new_value)
```

Note: CommitDatabase tracks history via mutations from CommitMutableState.

Committing

`commit_mutations()` returns the new commit id — capture it explicitly to chain further mutations or read the resulting state:

```
>>> commit_id = db.commit_mutations("Commit message", mutable_state)
```

Complete Example

Add an Alice document and read it back:

```
>>> key = TUTO_A_USER_LOGIN.create_key()
>>> login = TUTO_A_USER_LOGIN.create_document()
>>> login.nickname = "alice"
>>> login.password = "secret"

>>> mutable = CommitMutableState(db.initial_state())
>>> mutable.attachment_mutating().set(TUTO_A_USER_LOGIN, key, login)
>>> commit_id = db.commit_mutations("Add Alice", mutable)

>>> state = db.state(commit_id)
>>> state.attachment_getting().get(TUTO_A_USER_LOGIN, key)
Optional({nickname='alice', password='secret'})
```

Path-Based Mutators

Instead of replacing entire documents with `set()`, path-based mutators use **Paths** to target specific locations. This enables path-based merging when multiple users edit concurrently.

Mutator	Target	Operation
`update`	Field	Replace value at path
`union_in_set`	Set	Add elements
`subtract_in_set`	Set	Remove elements
`union_in_map`	Map	Add key-value pairs
`subtract_in_map`	Map	Remove keys
`update_in_map`	Map	Update existing key
`insert_in_xarray`	XArray	Insert at position
`update_in_xarray`	XArray	Update at position
`remove_in_xarray`	XArray	Remove at position

Field Update

```
>>> mutating.update(TUTO_A_USER_LOGIN, key, TUTO_P_LOGIN_NICKNAME, "alice_updated")
```

Set Operations

```
>>> mutating.union_in_set(GRAPH_A_TOPOLOGY, graph_key, GRAPH_P_TOPOLOGY_VERTEX_KEYS,
↪{new_vertex_key})

>>> mutating.subtract_in_set(GRAPH_A_TOPOLOGY, graph_key, GRAPH_P_TOPOLOGY_VERTEX_
↪KEYS, {deleted_vertex_key})
```

Map Operations

```
>>> mutating.union_in_map(attachment, key, path_to_map, {"new_key": "new_value"})
```

Why Paths Matter

When two users edit different fields simultaneously:

```
User A: update(attachment, key, path_to_name, "Alice")
User B: update(attachment, key, path_to_email, "bob@example.com")

After merge: Both updates apply (disjoint paths)
```

With `set()`, one user's changes would overwrite the other's.

Commit History

Inspect commit metadata:

```
>>> header = db.commit_header(commit_id)
>>> header.label()
'Add Alice'
>>> header.parent_commit_id() == ValueCommitId()
True
```

The first commit's parent is the zero `ValueCommitId` (no ancestor).

Navigate history by passing the explicit ids you captured:

```
>>> state1 = db.state(first_commit_id)
>>> state2 = db.state(latest_commit_id)
```

Embedded Definitions

CommitDatabase stores its definitions:

```
>>> defs = db.definitions()
>>> sorted(str(t) for t in defs.types())
['Tuto::Account', 'Tuto::Identity', 'Tuto::Login', 'Tuto::Status', 'Tuto::Texture',
↪'Tuto::Thumbnail', 'Tuto::User']
```

Calling `defs.inject()` makes `TUTO_A_USER_LOGIN`, `TUTO_S_LOGIN`, etc. available as constants in the calling namespace.

What's Next

- *Blobs* - Binary data storage
- *Serialization* - JSON and binary encoding

The Dual-Layer Contract

This page documents the contract between the Commit engine (exposed in Python through `dsviper`) and your application code. Read it before relying on commit behavior to validate your data: the engine produces **structurally sound but untrusted output**, and your application is what turns it into trusted state.

The Contract

Layer	Guarantees
Commit	Deterministic merge, DAG consistency, immutability
Application	Re-validates engine output before acting on it

When concurrent streams converge, mutations are applied using a **best-effort** algorithm:

- Mutations targeting non-existent documents or unresolved paths are **silently ignored**.
- Business rule violations are **not** detected by the engine.
- LWW arbitration may keep a value that no single submitted intent would have produced.

This is by design. The engine is intentionally agnostic to your domain rules and never refuses a merge — it picks a deterministic outcome and moves on.

Why the Engine's Output Is Untrusted Data

This is the load-bearing point of the contract.

"Untrusted" usually means *data that crossed a boundary you don't control* — network input, a deserialized file, an external API. The natural reflex is to validate at that boundary and then trust the data internally.

Best-effort merge breaks that reflex. **The state returned by `state(commitId)` is itself a boundary**, even though the data never left the process:

- Mutations you submitted may have been silently dropped.
- Two disjoint updates, each individually consistent, may have produced a combined state that violates a cross-field invariant.
- LWW may have preserved a value that no submitted commit would have written on its own.

Structurally the data is sound — types check, paths resolve, the DAG is consistent. **Semantically, you have no guarantee** that the result reflects any single coherent intent. That makes engine output equivalent to untrusted data from the application's perspective.

Three Error Families

Because of this, validation in a `dsviper`-based system happens at three distinct places, not one:

Family	Origin	Detected by
Untrusted (external)	I/O, deserialization, type mismatch, malformed path	Engine, fail-fast → `dsviper.Error`
Untrusted (post-merge)	State returned by the engine after convergence	Application, at read time
Invalid (semantic)	Business-rule violation on otherwise sound data	Application, at read time

The first family is what `dsviper.Error` covers. The other two are entirely your responsibility — and they share a remediation: **re-validate when you read the state, not when you build the mutations**.

What Commit Provides

Guarantee	Description
DAG Consistency	Commits form a valid directed acyclic graph
Immutability	Once committed, data cannot be modified
Deterministic Merge	Same inputs always produce the same output
Content-Addressable	`CommitId = SHA-1(content)`, tamper-evident

What Commit Does NOT Provide

Not Provided	Description
Intent Preservation	Deterministic arbitration (LWW), not your intent
Semantic Validation	No business rule checking
Mutation Notification	No alert when mutations are silently ignored
Conflict Detection	No notion of conflict — just deterministic merge

Implications for `dsviper` Code

The contract shapes how you should write code on top of `dsviper.Commit*`:

- **Treat the post-merge state as a boundary.** It is the symmetric equivalent of deserialized network input: structurally sound, semantically unverified.
- **Do not assume** that every operation you build into a `CommitMutableState` will land. After merging concurrent commits, some operations may have been silently dropped because their targets disappeared.
- **Validate at the application boundary,** which means the engine output. If your domain has invariants (uniqueness, referential integrity, cross-field consistency), enforce them when consuming the state.

Summary

> **"I guarantee structural integrity. I hand you back data that is structurally sound but semantically untrusted. You re-validate it on read. Together, we guarantee data integrity. Separately, neither of us can."**
>
> — The Dual-Layer Contract

See Also

- *Commit* — using the commit API from Python
- *Errors* — `dsviper.Error` and exception handling

Binary Data (Blobs)

Blobs provide efficient binary data storage with typed layouts and zero-copy NumPy integration.

When to use: Use blobs for binary data like images, meshes, and raw buffers. `BlobArray` for typed arrays, `BlobPack` for structured multi-region data.

Blob vs BlobId

Viper offers two DSM types for binary data:

Type	Use Case	Storage
`blob`	Small data (thumbnails, icons)	Inline in document
`blob_id`	Large data (textures, meshes)	Database blob API

blob (Inline)

A `blob` field stores binary data directly in the document. No special API needed.

blob_id (Reference)

A `blob_id` is a SHA-1 hash referencing a blob managed by the Database blob API. This requires the dedicated blob API:

```
>>> layout = BlobLayout()
>>> content = ValueBlob(bytes([1, 2, 3, 4, 5]))

>>> blob_id = db.create_blob(layout, content)
>>> blob_id
f07d73c81ed1a91165a75c5cc22253cc7895a8b2

>>> db.blob(blob_id)
blob(5)

>>> blob_id in db.blob_ids()
True

>>> info = db.blob_info(blob_id)
>>> info.size()
5
```

Content-addressable: The `blob_id` is computed from layout + content (SHA-1). Identical content always produces the same `blob_id`.

Constraint: A document referencing a `blob_id` cannot be committed unless the blob exists in the database. Always call `create_blob()` before using the `blob_id` in a document.

ValueBlob

A `ValueBlob` holds inline binary data. To pull the bytes back out, use the `bytes()` builtin (the buffer protocol) — there is no `.bytes()` method:

```
>>> data = bytes([1, 2, 3, 4, 5])
>>> blob = ValueBlob(data)

>>> bytes(blob)
b'\x01\x02\x03\x04\x05'

>>> len(blob)
5
```

ValueBlobId

A `ValueBlobId` references external binary data. The id is computed from the layout and content (deterministic SHA-1):

```
>>> layout = BlobLayout()
>>> content = ValueBlob(bytes([1, 2, 3, 4]))
>>> blob_id = ValueBlobId(layout, content)
>>> blob_id
6b8f3ca756046be29244d9bdb6b5ca5c00468ad5

>>> ValueBlobId.try_parse("6b8f3ca756046be29244d9bdb6b5ca5c00468ad5")
6b8f3ca756046be29244d9bdb6b5ca5c00468ad5
```

BlobLayout - Metadata Everywhere

A `BlobLayout` describes how to interpret blob bytes. This is the **Metadata Everywhere** principle applied to binary data: the layout is metadata that gives meaning to raw bytes.

```
>>> BlobLayout()
'uchar-1'

>>> BlobLayout('float', 3)
'float-3'

>>> BlobLayout('uint', 3)
'uint-3'

>>> BlobLayout('float', 2)
'float-2'
```

The layout enables:

- Type-safe interpretation of binary data
- Cross-platform compatibility (endianness handled)
- Validation at decode time

BlobView (Read-Only)

A `BlobView` interprets an existing blob with a given layout (read-only). When the layout has more than one component per element, indexing returns a tuple:

```
>>> import struct
>>> _buf = bytearray()
>>> for i in range(100):
...     _ = _buf.extend(struct.pack('<fff', float(i), float(i*2), float(i*3)))
>>> raw = ValueBlob(bytes(_buf))

>>> view = BlobView(BlobLayout('float', 3), raw)
>>> view.count()
100
>>> view[0]
(0.0, 0.0, 0.0)
>>> view[99]
(99.0, 198.0, 297.0)
```

Use `BlobView` when you need to read blob data without copying.

BlobArray (Read-Write)

A `BlobArray` is a typed array backed by a blob. Writes and reads use the NumPy buffer protocol — the array exposes a *flat* `(N*components,)` view, so reshape it to `(N, components)` to assign per-element tuples:

```
>>> import numpy as np
>>> layout = BlobLayout('float', 3)
>>> array = BlobArray(layout, 100)
```

(continues on next page)

(continued from previous page)

```
>>> np_view = np.array(array, copy=False).reshape(100, 3)
>>> np_view[0] = [1.0, 2.0, 3.0]
>>> np_view[1] = [4.0, 5.0, 6.0]

>>> view = BlobView(layout, array.blob())
>>> view[0]
(1.0, 2.0, 3.0)
>>> view[1]
(4.0, 5.0, 6.0)
```

BlobPack - Structured Binary Data

A `BlobPack` groups multiple named regions with different layouts into a single blob. This is ideal for complex structures like 3D meshes.

Example: 3D Mesh Storage

A mesh has positions, normals, UVs, and triangle indices — each with a different layout. Define the structure with a descriptor, then create the pack:

```
>>> descriptor = BlobPackDescriptor()
>>> descriptor.add_region('positions', BlobLayout('float', 3), 4)
>>> descriptor.add_region('normals', BlobLayout('float', 3), 4)
>>> descriptor.add_region('uvs', BlobLayout('float', 2), 4)
>>> descriptor.add_region('indices', BlobLayout('uint', 3), 2)

>>> mesh = BlobPack(descriptor)
>>> len(mesh)
4
```

Fill the Mesh Data

Each region exposes a flat NumPy view; reshape to write per-vertex tuples:

```
>>> import numpy as np
>>> pos = np.array(mesh['positions'], copy=False).reshape(4, 3)
>>> pos[:] = [[-1.0, -1.0, 0.0], [1.0, -1.0, 0.0],
...           [ 1.0,  1.0, 0.0], [-1.0,  1.0, 0.0]]

>>> normals = np.array(mesh['normals'], copy=False).reshape(4, 3)
>>> normals[:] = [[0.0, 0.0, 1.0]] * 4

>>> uvs = np.array(mesh['uvs'], copy=False).reshape(4, 2)
>>> uvs[:] = [[0.0, 0.0], [1.0, 0.0], [1.0, 1.0], [0.0, 1.0]]

>>> indices = np.array(mesh['indices'], copy=False).reshape(2, 3)
>>> indices[:] = [[0, 1, 2], [0, 2, 3]]
```

Serialize and Restore

```
>>> blob = mesh.blob()
>>> restored = BlobPack.from_blob(blob)

>>> np.array(restored['positions'], copy=False).reshape(4, 3)[0].tolist()
[-1.0, -1.0, 0.0]
>>> np.array(restored['indices'], copy=False).reshape(2, 3)[1].tolist()
[0, 2, 3]
```

Region Access

```
>>> 'positions' in mesh
True
>>> 'colors' in mesh
False

>>> mesh['positions'].name()
'positions'
>>> mesh['positions'].count()
4
>>> mesh['positions'].blob_layout()
'float-3'
>>> mesh['positions'].data_count()
12
>>> mesh['positions'].byte_count()
48
```

Why BlobPack?

Benefit	Description
Single blob	All mesh data in one blob_id
Typed regions	Each region has its own layout
Self-describing	Layout metadata embedded in header
Efficient	Direct memory mapping, no parsing

NumPy Integration

`BlobArray` implements the Python Buffer Protocol, enabling zero-copy interoperability with NumPy and other array libraries.

Zero-Copy View

The buffer is exposed as a flat 1D array of components — reshape to give it the geometric shape you want:

```
>>> import numpy as np
>>> layout = BlobLayout('float', 3)
>>> positions = BlobArray(layout, 100)

>>> np_view = np.array(positions, copy=False)
```

(continues on next page)

(continued from previous page)

```
>>> np_view.shape
(300,)
>>> np_view.dtype
dtype('float32')

>>> np_view.reshape(100, 3).shape
(100, 3)
```

Bidirectional Modifications

Changes through NumPy affect the original BlobArray:

```
>>> reshaped = np.array(positions, copy=False).reshape(100, 3)
>>> reshaped[0] = [10.0, 20.0, 30.0]

>>> view = BlobView(layout, positions.blob())
>>> view[0]
(10.0, 20.0, 30.0)
```

Direct Memory Access

```
>>> mv = memoryview(positions)
>>> mv.nbytes
1200
```

BlobPack Regions

`BlobPackRegion` also supports the Buffer Protocol — same flat-then-reshape idiom:

```
>>> positions_np = np.array(mesh['positions'], copy=False).reshape(4, 3)
>>> normals_np = np.array(mesh['normals'], copy=False).reshape(4, 3)
>>> uvs_np = np.array(mesh['uvs'], copy=False).reshape(4, 2)

>>> positions_np.shape
(4, 3)
>>> uvs_np.shape
(4, 2)

>>> positions_np *= 2.0
>>> positions_np[0].tolist()
[-2.0, -2.0, 0.0]
```

Why Zero-Copy Matters

Scenario	Without Zero-Copy	With Zero-Copy
1M vertices	Copy 12MB	Share pointer
GPU upload	Python → copy → C++	Direct access
Scientific compute	Data duplication	In-place ops

Blob in Attachments

Blobs can be used in attachments:

```
// Small data inline
struct Thumbnail {
uint16 width;
uint16 height;
blob data;
// Inline binary
};

// Large data by reference
struct Texture {
uint16 width;
uint16 height;
blob_id pixels;
// External reference
};
```

Storing Blobs in Database

Inline Blobs

Inline blobs are stored directly in the document. The Tuto fixture exposes a `Thumbnail` struct with a `blob` field, attached to `User` as `avatar`:

```
>>> user_key = TUTO_A_USER_AVATAR.create_key()

>>> thumb = TUTO_A_USER_AVATAR.create_document()
>>> thumb.width = 64
>>> thumb.height = 64
>>> thumb.data = ValueBlob(bytes([1, 2, 3, 4, 5]))
>>> thumb
{width=64, height=64, data=blob(5)}

>>> ms = CommitMutableState(db.initial_state())
>>> ms.attachment_mutating().set(TUTO_A_USER_AVATAR, user_key, thumb)
>>> avatar_commit = db.commit_mutations("Add avatar", ms)

>>> db.state(avatar_commit).attachment_getting().get(TUTO_A_USER_AVATAR, user_key)
Optional({width=64, height=64, data=blob(5)})
```

Referenced Blobs (blob_id)

Use the database blob API to store and retrieve. The Tuto fixture exposes a `Texture` struct with a `blob_id` field, attached to `User` as `portrait`:

```
>>> mesh_layout = BlobLayout()
>>> mesh_content = ValueBlob(bytes([10, 20, 30, 40]))
>>> texture_blob_id = db.create_blob(mesh_layout, mesh_content)

>>> texture = TUTO_A_USER_PORTRAIT.create_document()
```

(continues on next page)

(continued from previous page)

```
>>> texture.width = 1024
>>> texture.height = 1024
>>> texture.pixels = texture_blob_id

>>> ms = CommitMutableState(db.state(avatar_commit))
>>> ms.attachment_mutating().set(TUTO_A_USER_PORTRAIT, user_key, texture)
>>> portrait_commit = db.commit_mutations("Add portrait", ms)

>>> db.state(portrait_commit).attachment_getting().get(TUTO_A_USER_PORTRAIT, user_key)
Optional({width=1024, height=1024, pixels=...})
```

Retrieving Blobs

```
>>> stored_id = db.create_blob(BlobLayout(), ValueBlob(bytes([10, 20, 30, 40, 50])))
>>> bytes(db.blob(stored_id))
b'\n\x14\x1e(2'

>>> db.read_blob(stored_id, size=2, offset=0)
blob(2)
```

BlobStream (Large Blobs)

For very large blobs, use streaming to avoid loading everything in memory.

Required for blobs > 2GB: The standard `create_blob()` API has a 2GB size limit. Use BlobStream for larger data:

```
>>> stream = db.blob_stream_create(BlobLayout('uchar', 1), size=10)

>>> db.blob_stream_append(stream, ValueBlob(bytes([1, 2, 3, 4, 5])))
>>> db.blob_stream_append(stream, ValueBlob(bytes([6, 7, 8, 9, 10])))

>>> stream_blob_id = db.blob_stream_close(stream)
>>> bytes(db.blob(stream_blob_id))
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n'
```

This is essential for:

- 3D meshes with millions of vertices
- Video/audio data

When to Use Each Type

Scenario	Recommendation
Thumbnails (< 64KB)	Use `blob` (inline)
Textures (> 1MB)	Use `blob_id` (Database blob API)
Mesh geometry	Use `blob_id` (Database blob API)
Icons, small images	Use `blob` (inline)
Audio/video	Use `blob_id` (Database blob API)

What's Next

- *Serialization* - JSON and binary encoding

Serialization

Viper provides multiple serialization formats for values and definitions. This chapter covers JSON and binary encoding.

JSON Encoding

Encoding Values

Encode any value to JSON:

```
>>> from dsviper import *

>>> v = Value.deduce([1, 2, 3])
>>> Value.json_encode(v)
'[1,2,3]'
```

Complex values — Python tuples become JSON arrays:

```
>>> v = Value.deduce({(1, 2): "one", (3, 4): "two"})
>>> Value.json_encode(v)
'[[[1,2],"one"],[[3,4],"two"]]'
```

Decoding Values

Decode JSON back to a value:

```
>>> json_str = '[1,2,3]'
>>> t = TypeVector(Type.INT64)
>>> v = Value.json_decode(json_str, t, Definitions().const())
>>> v
[1, 2, 3]
```

Pretty Printing

Define a small `Login` structure to demonstrate pretty printing:

```
>>> defs = Definitions()
>>> ns = NameSpace(ValueUUId("f529bc42-0618-4f54-a3fb-d55f95c5ad03"), "Tuto")
>>> desc = TypeStructureDescriptor("Login")
>>> desc.add_field("nickname", Type.STRING)
>>> desc.add_field("password", Type.STRING)
>>> t_login = defs.create_structure(ns, desc)
```

Format JSON with indentation:

```
>>> v = Value.create(t_login, {"nickname": "alice", "password": "secret"})
>>> print(Value.json_encode(v, indent=2))
{
  "nickname": "alice",
```

(continues on next page)

(continued from previous page)

```
  "password": "secret"
}
```

DSM Definitions Serialization

To JSON

Convert `Definitions` to a `DSMDefinitions` snapshot, then encode to JSON:

```
>>> defs2 = Definitions()
>>> ns2 = NameSpace(ValueUUId("aaaaaaaa-0000-0000-0000-000000000001"), "Demo")
>>> defs2.create_concept(ns2, "User")
Demo::User

>>> dsm_defs = defs2.const().to_dsm_definitions()
>>> json_str = dsm_defs.json_encode(indent=2)
>>> "Demo" in json_str and "User" in json_str
True
```

The encoded JSON contains the standard DSM categories — namespaces, concepts, structures, enumerations, attachments, clubs, function pools, and attachment function pools.

From JSON

```
>>> restored = DSMDefinitions.json_decode(json_str)
>>> type(restored).__name__
'DSMDefinitions'
```

To DSM Language

Render definitions back to DSM source:

```
>>> print(dsm_defs.to_dsm())
namespace Demo {aaaaaaaa-0000-0000-0000-000000000001} {

concept User;

}; // ns Demo
```

Binary Encoding

Binary encoding is compact and fast, used for persistence and network transfer.

Encoding Values

`Value.encode(value)` returns a `ValueBlob`; `Value.decode(blob, type, defs)` restores the value:

```
>>> v = Value.deduce([1, 2, 3])
>>> blob = Value.encode(v)
>>> blob
blob(...)
```

(continues on next page)

(continued from previous page)

```
>>> restored = Value.decode(blob, v.type(), Definitions().const())
>>> restored == v
True
```

Encoding Definitions

`Definitions` are encoded via the const view:

```
>>> blob = defs2.const().encode()
>>> blob
blob(...)

>>> restored = Definitions.decode(blob)
>>> type(restored).__name__
'Definitions'
```

Stream Codec

For custom binary formats and RPC, Viper provides low-level stream codecs that encode/decode primitive types directly.

Available Codecs

Codec	Description
`Codec.STREAM_BINARY`	Compact binary, no type checking
`Codec.STREAM_TOKEN_BINARY`	Adds type tokens for validation

The token variant detects desynchronization early by prefixing each value with its type.

Encoding

Create an encoder, write primitives, finalize:

```
>>> encoder = Codec.STREAM_BINARY.create_encoder()
>>> encoder.write_int64(42)
>>> encoder.write_float(3.14)
>>> encoder.write_string("hello")

>>> blob = encoder.end_encoding()
>>> blob
blob(...)
```

Decoding

Read in the same order. Note that `write_float` / `read_float` round-trip through 32-bit float, so `3.14` comes back with float32 precision:

```
>>> decoder = Codec.STREAM_BINARY.create_decoder(blob)
>>> decoder.read_int64()
```

(continues on next page)

(continued from previous page)

```
42
>>> decoder.read_float()
3.140000104904175
>>> decoder.read_string()
'hello'

>>> decoder.has_more()
False
```

Type Safety with Tokens

The token codec rejects mismatched reads:

```
>>> encoder = Codec.STREAM_TOKEN_BINARY.create_encoder()
>>> encoder.write_int64(42)
>>> blob = encoder.end_encoding()

>>> decoder = Codec.STREAM_TOKEN_BINARY.create_decoder(blob)
>>> decoder.read_bool()
Traceback (most recent call last):
    ...
dsviper.ViperError: ...Expected token bool, got int64...
```

Computing Sizes

```
>>> sizer = Codec.STREAM_BINARY.create_sizer()
>>> sizer.size_of_float()
4
>>> sizer.size_of_int64()
8
```

Value Description

Get a descriptive string representation:

```
>>> v = Value.deduce([1, 2, 3])
>>> v.description()
'[1, 2, 3]:vector<int64>'

>>> login = Value.create(t_login, {"nickname": "alice"})
>>> login.description()
"{nickname='alice':string, password='':string}:Tuto::Login"
```

Binary File Format

DSM definitions can be saved in binary format (`.dsmb`):

```
# Write definitions to binary file
>>> with open("model.dsmb", "wb") as f:
...     f.write(defs.const().encode())
```

(continues on next page)

(continued from previous page)

```
# Read definitions from binary file
>>> with open("model.dsmb", "rb") as f:
...     defs = Definitions.decode(f.read())
```

Common Patterns

Round-Trip Test

Verify encoding/decoding preserves data:

```
>>> original = Value.create(t_login, {"nickname": "test"})
>>> json_str = Value.json_encode(original)
>>> decoded = Value.json_decode(json_str, t_login, defs.const())
>>> original == decoded
True
```

Schema Export

Export schema for external systems:

```
>>> defs = db.definitions()
>>> dsm_defs = defs.to_dsm_definitions()
>>> print(dsm_defs.to_dsm())           # DSM language
>>> print(dsm_defs.json_encode(indent=2))   # JSON schema
```

Note

The `Schema Export` snippet above pulls definitions from a live `db` and therefore requires a working database. See *Database* for a full walkthrough.

Summary

Format	Use Case	API
JSON	Debugging, interop	`Value.json_encode/decode`
Binary	Persistence, RPC	`Value.encode/decode`
DSM	Human-readable schema	`dsm_defs.to_dsm()`

What's Next

- *DSM Processing* - Loading and introspecting DSM files
- *DSM Manual* - Data modeling language
- *Toolchain* - Development tools

3.3 API Reference

This reference documents all public classes in the `dsviper` module, organized by functional domain.

3.3.1 Type System

The type system defines all available types in Viper. `Type` is the base class with static accessors for primitive types.

When to use: Use types to define schemas, validate data, and create parameterized containers like vectors and maps.

Quick Start

```
from dsviper import Type, TypeVector, TypeMap, TypeOptional

# Primitive type constants
t_int = Type.INT64
t_str = Type.STRING

# Parameterized container types
t_vec = TypeVector(Type.STRING)           # list of strings
t_map = TypeMap(Type.STRING, Type.INT64)  # dict[str, int]

# Nested types
t_nested = TypeMap(Type.STRING, TypeVector(Type.FLOAT))

# Nullable type
t_opt = TypeOptional(Type.STRING)
```

Choosing the Right Type

Use Case	Type Class	Example
Primitive value	`Type.STRING`, `Type.INT64`	Built-in constants
Homogeneous list	`TypeVector`	`TypeVector(Type.INT64)`
Key-value mapping	`TypeMap`	`TypeMap(Type.STRING, Type.INT64)`
Nullable value	`TypeOptional`	`TypeOptional(Type.STRING)`
Concurrent list	`TypeXArray`	`TypeXArray(Type.STRING)`
Fixed-size tuple	`TypeTuple`	`TypeTuple([Type.INT64, Type.STRING])`

Base Class

dsviper.Type	A utility class to handle type representation.
dsviper.TypeName	A class used to describe the name of a type.

Type

class dsviper.**Type**

Bases: `object`

A utility class to handle type representation.

Note: Not directly instantiable.

ANY = any

ANY_CONCEPT = any_concept

BLOB = blob

BLOB_ID = blob_id

BOOL = bool

COMMIT_ID = commit_id

DOUBLE = double

FLOAT = float

INT16 = int16

INT32 = int32

INT64 = int64

INT8 = int8

STRING = string

UINT16 = uint16

UINT32 = uint32

UINT64 = uint64

UINT8 = uint8

UUID = uuid

VOID = void

static decode(*blob:* ValueBlob, *definitions:* DefinitionsConst, * *(Keyword-only parameters separator (PEP 3102))*, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *Type*

Return a type by decoding the blob with a StreamBinaryCodec if not specified.

static encode(*type:* Type, *, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *ValueBlob*

Return a blob that encodes the type with a StreamBinaryCodec if not specified.

static hexdigest(*type:* Type, *, *hashing:* Hashing | *None* = *None*) → str

Return the hexa digest of the type computed with the hashing interface if specified else use SHA1.

static is_sized(*type:* Type) → bool

Return True if the type has a fixed size.

static read(*stream_reading:* StreamReading, *definitions:* DefinitionsConst) → *Type*

Read and return a type.

static size_of(*type:* Type, *stream_sizing:* StreamSizing) → int

Return the fixed size of the type or raise.

static use_blob_id(*type:* Type) → bool

Return True if the type uses the type blob_id.

static write(*type:* Type, *stream_writing:* StreamWriting) → None

Write a type.

TypeName

class dsviper.**TypeName**

Bases: object

A class used to describe the name of a type.

Note: Not directly instantiable.

name() → str

Return the name.

name_space() → *NameSpace*

Return the namespace.

Primitive Types

dsviper.TypeVoid	A class used to represent the void type.
dsviper.TypeBool	A class used to represent the bool type.
dsviper.TypeUInt8	A class used to represent the uint8 type.
dsviper.TypeUInt16	A class used to represent the uint16 type.
dsviper.TypeUInt32	A class used to represent the uint32 type.
dsviper.TypeUInt64	A class used to represent the uint64 type.
dsviper.TypeInt8	A class used to represent the int8 type.
dsviper.TypeInt16	A class used to represent the int16 type.
dsviper.TypeInt32	A class used to represent the int32 type.
dsviper.TypeInt64	A class used to represent the int64 type.
dsviper.TypeFloat	A class used to represent the float type.
dsviper.TypeDouble	A class used to represent the double type.
dsviper.TypeString	A class used to represent the string type.
dsviper.TypeBlob	A class used to represent the blob type.
dsviper.TypeBlobId	A class used to represent the blob_id type.
dsviper.TypeCommitId	A class used to represent the commit_id type.
dsviper.TypeUUId	A class used to represent the uuid type.

TypeVoid

class dsviper.**TypeVoid**

Bases: object

A class used to represent the void type.

Use the static property Type.VOID.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeBool

class dsviper.**TypeBool**

Bases: object

A class used to represent the bool type.

Use the static property Type.BOOL.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeUInt8

class dsviper.**TypeUInt8**

Bases: object

A class used to represent the uint8 type.

Use the static property Type.UINT8.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeUInt16

class dsviper.**TypeUInt16**

Bases: object

A class used to represent the uint16 type.

Use the static property Type.UINT16.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeUInt32

class dsviper.**TypeUInt32**

Bases: object

A class used to represent the uint32 type.

Use the static property Type.UINT32.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeUInt64

class dsviper.**TypeUInt64**

Bases: object

A class used to represent the uint64 type.

Use the static property Type.UINT64.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeInt8

class dsviper.**TypeInt8**
Bases: object

A class used to represent the int8 type.

Use the static property Type.INT8.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeInt16

class dsviper.**TypeInt16**
Bases: object

A class used to represent the int16 type.

Use the static property Type.INT16.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeInt32

class dsviper.**TypeInt32**

Bases: object

A class used to represent the int32 type.

Use the static property Type.INT32.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeInt64

class dsviper.**TypeInt64**

Bases: object

A class used to represent the int64 type.

Use the static property Type.INT64.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return A TypeName.

TypeFloat

class dsviper.**TypeFloat**

Bases: object

A class used to represent the float type.

Use the static property Type.FLOAT.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeDouble

class dsviper.**TypeDouble**
Bases: object

A class used to represent the double type.

Use the static property Type.DOUBLE.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeString

class dsviper.**TypeString**
Bases: object

A class used to represent the string type.

Use the static property Type.STRING.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeBlob

class dsviper.**TypeBlob**

Bases: object

A class used to represent the blob type.

Use the static property Type.BLOB.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeBlobId

class dsviper.**TypeBlobId**

Bases: object

A class used to represent the blob_id type.

Use the static property Type.BLOB_ID.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeCommitId

class dsviper.**TypeCommitId**

Bases: object

A class used to represent the commit_id type.

Use the static property Type.COMMIT_ID.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeUUId

class dsviper.**TypeUUId**

Bases: object

A class used to represent the uuid type.

Use the static property Type.UUID.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return A TypeName.

Container Types

dsviper.TypeVector	A class used to represent a vector<element_type> type.
dsviper.TypeSet	A class used to represent a set<element_type> type.
dsviper.TypeMap	A class used to represent a map<element_type> type.
dsviper.TypeXArray	A class used to represent a xarray<element_type> type.
dsviper.TypeOptional	A class used to represent an optional<element_type> type.

TypeVector

class dsviper.**TypeVector**(*element_type*)

Bases: object

A class used to represent a vector<element_type> type.

static cast(*type:* Type) → *TypeOptional*

Return a type vector<element_type> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

element_type() → *Type*
Return the element type.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

TypeSet

class dsviper.**TypeSet**(*element_type*)
Bases: object

A class used to represent a set<element_type> type.

static cast(*type:* Type) → *TypeSet*
Return the type set<element_type> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

element_type() → *Type*
Return the element type.

elements_type() → *TypeVector*
Return the type for elements.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

TypeMap

class dsviper.**TypeMap**(*key_type*, *element_type*)
Bases: object

A class used to represent a map<element_type> type.

static cast(*type:* Type) → *TypeMap*
Return the type map<key_type, element_type> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

element_type() → *Type*
Return the element type.

item_type() → *TypeTuple*
Return the type for item.

items_type() → *TypeVector*
Return the type for items.

key_type() → *Type*
Return the KeyType.

keys_type() → *TypeSet*
Return the type for keys.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

values_type() → *TypeVector*
Return the type for values.

TypeXArray

class dsviper.**TypeXArray**(*element_type*)
Bases: object

A class used to represent a xarray<element_type> type.

static cast(*type:* Type) → *TypeXArray*
Return a type xarray<element_type> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

element_type() → *Type*
Return the type.

elements_type() → *TypeVector*
Return the type for elements.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

TypeOptional

class dsviper.**TypeOptional**(*element_type*)
Bases: object

A class used to represent an optional<element_type> type.

static cast(*type:* Type) → *TypeOptional*

Return the type optional<element_type> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

element_type() → *Type*

Return the element type.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

Algebraic Types

dsviper.TypeTuple	A class used to represent a tuple<T0, ...> type.
dsviper.TypeVec	A class used to represent a vec<element_type, size> type.
dsviper.TypeMat	A class used to represent a mat<numeric_type, columns, rows> type.
dsviper.TypeVariant	A class used to represent a variant<T0, ...> type.
dsviper.TypeAny	A class used to represent the any type.

TypeTuple

class dsviper.**TypeTuple**(*types*)

Bases: object

A class used to represent a tuple<T0, …> type.

static cast(*type:* Type) → *TypeTuple*

Return a type tuple<T0, …> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

types() → list[*Type*]

Return the list of types.

TypeVec

class dsviper.**TypeVec**(*element_type*, *size*)

Bases: object

A class used to represent a vec<element_type, size> type.

static cast(*type:* Type) → *TypeVec*

Return a vec<N, size> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

element_type() → _NumericTypes

Return the element type.

elements_type() → *TypeVector*

Return the type for elements.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

size() → int

Return the number of elements.

type_code() → str

Return the type code.

TypeMat

class dsviper.**TypeMat**(*numeric_type*, *columns*, *rows*)

Bases: object

A class used to represent a mat<numeric_type, columns, rows> type.

static cast(*type:* Type) → *TypeMat*

Return the type mat<N, columns, rows> or raise.

columns() → int

Return the number of columns.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

element_type() → _NumericTypes

Return the element type.

elements_type() → *TypeVector*

Return the type for elements.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

rows() → int

Return the number of rows.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

TypeVariant

class dsviper.**TypeVariant**(*types*)

Bases: object

A class used to represent a variant<T0, …> type.

static cast(*type:* Type) → *TypeVariant*

Return a type variant<T0, …> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

types() → list[*Type*]

Return the list of types.

TypeAny

class dsviper.**TypeAny**

Bases: object

A class used to represent the any type.

Use the static property Type.ANY.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

User-Defined Types

dsviper.TypeStructure	A class used to represent a struct type.
dsviper.TypeStructureDescriptor	A class used to describe the fields of a struct.
dsviper.TypeStructureField	A class used to represent a field for a struct.
dsviper.TypeEnumeration	A class used to represent an enum type.
dsviper.TypeEnumerationCase	A class used to represent a case for an enum type.
dsviper.TypeEnumerationDescriptor	A class used to describe the cases of an enum.

TypeStructure

class dsviper.**TypeStructure**

Bases: object

A class used to represent a struct type.

Note: Not directly instantiable.

static cast(*type:* Type) → *TypeStructure*

Return a type struct or raise.

check(*field_name: str*) → *TypeStructureField*

Return the field or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

documentation() → str

Return The documentation.

fields() → list[*TypeStructureField*]

Return the list of fields.

is_compact() → bool

Return True the type is compact.

query(*field_name: str*) → *TypeStructureField* | None

Return the field or None.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeStructureDescriptor

class dsviper.**TypeStructureDescriptor**(*name*, *, *documentation=None*)

Bases: object

A class used to describe the fields of a struct.

add_field(*field_name: str*, *type_or_value: _DefaultValues*, *documentation: str | None = None*) → None
Add a field where the type is specified or deducted from the value.

description() → str
Return the description.

documentation() → str
Return the documentation.

fields() → list[*TypeStructureField*]
Return the list of fields.

name() → str
Return the name.

TypeStructureField

class dsviper.**TypeStructureField**
Bases: object

A class used to represent a field for a struct.

Note: Not directly instantiable.

default_value() → *Value* | None
Return the default value or None.

documentation() → str
Return the documentation.

name() → str
Return the name.

type() → *Type*
Return the type.

TypeEnumeration

class dsviper.**TypeEnumeration**
Bases: object

A class used to represent an enum type.

Note: Not directly instantiable.

cases() → list[*TypeEnumerationCase*]
Return the list of cases.

static cast(*type:* Type) → *TypeEnumeration*
Return a type enum or raise.

check(*name: str*) → *TypeEnumerationCase*
Return the case or raise.

description(*, *namespace:* NameSpace | *None = None*) → str
Return the description.

documentation() → str

Return the documentation.

query(*name: str*) → *TypeEnumerationCase* | None

Return the case or None.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeEnumerationCase

class dsviper.**TypeEnumerationCase**

Bases: object

A class used to represent a case for an enum type.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

name() → str

Return the name.

TypeEnumerationDescriptor

class dsviper.**TypeEnumerationDescriptor**(*name*, *, *documentation=None*)

Bases: object

A class used to describe the cases of an enum.

add_case(*name: str*, *documentation: str* | *None* = *None*)

Add a case to the enum.

cases() → list[*TypeEnumerationCase*]

Return the list of cases.

description() → str

Return the description.

documentation() → str

Return the documentation.

name() → str

Return the name.

Concept Types

dsviper.TypeConcept	A class used to represent a concept type.
dsviper.TypeClub	A class used to represent a club type.
dsviper.TypeKey	A class used to represent a key<element_type> type where element_type is a concept, club or any_concept.
dsviper.TypeAnyConcept	A class used to represent the any_concept type.

TypeConcept

class dsviper.**TypeConcept**

Bases: object

A class used to represent a concept type.

Note: Not directly instantiable.

static cast(*type:* Type) → *TypeConcept*

Return a concept or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

documentation() → str

Return the documentation.

is_member(*type_concept:* TypeConcept) → bool

Return True if type_concept is related.

parent() → *TypeConcept* | None

Return the parent concept or None.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

TypeClub

class dsviper.**TypeClub**

Bases: object

A class used to represent a club type.

Note: Not directly instantiable.

static cast(*type:* Type) → *TypeClub*

Return a type club or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

documentation() → str
Return the documentation.

is_member(*type_concept:* TypeConcept) → bool
Return True type_concept is a member.

members() → list[*TypeConcept*]
Return the list of concepts.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

type_name() → *TypeName*
Return a TypeName.

TypeKey

class dsviper.**TypeKey**(*element_type*)
Bases: object

A class used to represent a key<element_type> type where element_type is a concept, club or any_concept.

static cast(*type:* Type) → *TypeKey*
Return the type key<element_type> or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

element_type() → *Type*
Return the element type.

is_any_concept() → bool
Return True if element_type is any_concept.

is_club() → bool
Return True if element_type is a club.

is_concept() → bool
Return True if element_type is a concept.

representation(*, *namespace:* NameSpace | *None* = *None*) → str
Return the representation.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_code() → str
Return the type code.

TypeAnyConcept

class dsviper.**TypeAnyConcept**

Bases: object

A class used to represent the any_concept type.

Use the static property Type.ANY_CONCEPT.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_code() → str

Return the type code.

type_name() → *TypeName*

Return a TypeName.

3.3.2 Values

Values are instances of types. `Value` is the base class with factory methods for creating and converting values.

When to use: Use values to hold typed data. Primitives are immutable; containers (Vector, Map, Set, XArray) are mutable.

Quick Start

```
from dsviper import Value, ValueString, ValueInt64, ValueVector, TypeVector, Type

# Direct construction for primitives
name = ValueString("Alice")
count = ValueInt64(42)

# Factory method with type (for containers)
t_vec = TypeVector(Type.INT64)
numbers = Value.create(t_vec, [1, 2, 3])

# Access underlying Python value
print(name.get())          # "Alice"
print(list(numbers))       # [1, 2, 3]

# Check type
print(name.type())         # string
```

Choosing the Right Pattern

Pattern	When to Use	Example
`ValueString("...")`	Direct primitive construction	`ValueInt64(42)`
`Value.create(type, data)`	Generic factory with type	`Value.create(t_vec, [1,2,3])`
`Value.decode(type, bytes)`	Deserialize from binary	`Value.decode(Type.STRING, data)`
`ValueXxx.cast(value)`	Type casting	`ValueInt64.cast(v)`

Base Class

dsviper.Value	A utility class to handle value instantiation and representation.

Value

class dsviper.**Value**

Bases: `object`

A utility class to handle value instantiation and representation.

Note: Not directly instantiable.

static bson_decode(*blob:* ValueBlob, *type_structure:* TypeStructure, *definitions:* DefinitionsConst) → *ValueStructure*

Return a value by decoding the bson encoded blob of a structure.

static bson_encode(*value:* ValueStructure) → *ValueBlob*

Return the bson encoded blob of the value.

static collect_blob_ids(*object:* Value) → set[*ValueBlobId*]

Return the set of blob_id of all referenced blob_id by the object, where object must be a Value, a Path, a CommitState, a CommitMutableState or a ValueProgram.

static copy(*value:* Value) → *Value*

Return a copy of a value (deepcopy).

static create(*type:* Type, *initial_value: _InputValues | None = None*) → *Value*

Return an initialized value from the type.

static decode(*blob:* ValueBlob, *type:* Type, *definitions:* DefinitionsConst, *, *stream_codec_instancing:* StreamCodecInstancing | *None = None*, *pack_sized: bool = False*) → *Value*

Return a Value by decoding the blob with a StreamBinaryCodec if not specified.

If pack_sized is True, then the vector<T> where T is a Sized type keeps an element in the binary encoded representation. This feature is only used for decoding huge immutable resource by deferred the decoding cost to the method ValueVector.at(index).

static deduce(*value: _PythonValues*) → *Value*

Return a strong typed conversion of a Python object.

static dumps(*value:* Value, *, *json: bool = False*) → _PythonValues

Return a projection to Python objects of the strong typed value. If json is True then: - a set is converted to a list. - a map is converted to a list of (key, value).

static encode(*value:* Value, *, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *ValueBlob*

Return a blob that encodes the value with a StreamBinaryCodec if not specified.

static hexdigest(*value:* Value, *, *hashing:* Hashing | *None* = *None*) → str

Hash the value with the hashing interface if specified else use SHA1.

static json_decode(*string: str*, *type:* Type, *definitions:* DefinitionsConst) → *Value*

Return a value by decoding the json encoded string.

static json_encode(*value:* Value, *indent: int = -1*) → str

Return the JSON encoded string for the value.

static loads(*object: _PythonValues*, *type:* Type, *definitions:* DefinitionsConst) → *Value*

Return a value by decoding the Python object.

static read(*type:* Type, *stream_reading:* StreamReading, *definitions:* DefinitionsConst, *, *pack_sized: bool = False*) → *Value*

Read the value from the stream.

static succ(*value:* Value) → *Value*

Return the successor of a value.

static write(*value:* Value, *stream_writing:* StreamWriting) → None

Write the value to the stream.

Primitive Values

dsviper.ValueVoid	A class used to represent a value of type void.
dsviper.ValueBool	A class used to represent a value of type bool.
dsviper.ValueUInt8	A class used to represent a value of type uint8.
dsviper.ValueUInt16	A class used to represent a value of type uint16.
dsviper.ValueUInt32	A class used to represent a value of type uint32.
dsviper.ValueUInt64	A class used to represent a value of type uint64.
dsviper.ValueInt8	A class used to represent a value of type int8.
dsviper.ValueInt16	A class used to represent a value of type int16.
dsviper.ValueInt32	A class used to represent a value of type int32.
dsviper.ValueInt64	A class used to represent a value of type int64.
dsviper.ValueFloat	A class used to represent a value of type float.
dsviper.ValueDouble	A class used to represent a value of type double.
dsviper.ValueString	A class used to represent a value of type string.
dsviper.ValueBlob	A class used to represent a value of type blob.
dsviper.ValueBlobId	A class used to represent a value of type blob_id.
dsviper.ValueCommitId	A class used to represent a value of type commit_id.
dsviper.ValueUUId	A class used to represent a value of type uuid.

ValueVoid

class dsviper.**ValueVoid**

Bases: object

A class used to represent a value of type void. Seamless with Python None. Or use Value.create(Type.VOID).

Note: Not directly instantiable.

static cast(*value:* Value) → *ValueVoid*

Return void or raise.

copy() → *ValueVoid*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → *Value*

Return Python None.

hash() → int

Return the hash value.

representation() → str

Return the representation.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueBool

class dsviper.**ValueBool**(*value*)

Bases: object

A class used to represent a value of type bool. Seamless with a Python bool.

Or use Value.create(Type.BOOL [, TruelFalse]).

FALSE = false

TRUE = true

static cast(*value:* Value) → *ValueBool*

Return a bool or raise.

copy() → *ValueBool*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → bool

Return a Python bool.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueBool* | None

Return a bool or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueUInt8

class dsviper.**ValueUInt8**(*initial_value=None*)

Bases: object

A class used to represent a value of type uint8. Seamless with a Python int.

Or use Value.create(Type.UINT8 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueUInt8*

Return an uint8 or raise.

copy() → *ValueUInt8*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → int

Return a Python int.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueUInt8* | None

Return an uint8 or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueUInt16

class dsviper.**ValueUInt16**(*initial_value=None*)

Bases: object

A class used to represent a value of type uint16. Seamless with a Python int.

Or use Value.create(Type.UINT16 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueUInt16*

Return an uint16 or raise.

copy() → *ValueUInt16*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → int

Return a Python int.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueUInt16* | None

Return an uint16 or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueUInt32

class dsviper.**ValueUInt32**(*initial_value=None*)

Bases: object

A class used to represent a value of type uint32. Seamless with a Python int.

Or use Value.create(Type.UINT32 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueUInt32*

Return an uint32 or raise.

copy() → *ValueUInt32*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → int

Return a Python int.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueUInt32* | None
Return an uint32 or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueUInt64

class dsviper.**ValueUInt64**(*initial_value=None*)
Bases: object

A class used to represent a value of type uint64. Seamless with a Python int.

Or use Value.create(Type.UINT64 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueUInt64*
Return an uint64 or raise.

copy() → *ValueUInt64*
Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

encoded() → int
Return a Python int.

hash() → int
Return the hash value.

representation() → str
Return the representation.

static try_parse(*string: str*) → *ValueUInt64* | None
Return an uint64 or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueInt8

class dsviper.**ValueInt8**(*initial_value=None*)
Bases: object

A class used to represent a value of type int8. Seamless with a Python int.

Or use Value.create(Type.INT8 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueInt8*

Return an int8 or raise.

copy() → *ValueInt8*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → int

Return a Python int.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueInt8* | None

Return an int8 or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueInt16

class dsviper.**ValueInt16**(*initial_value=None*)

Bases: object

A class used to represent a value of type int16. Seamless with a Python int.

Or use Value.create(Type.INT16 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueInt16*

Return an int16 or raise.

copy() → *ValueInt16*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → int

Return a Python int.

hash() → int

Return the hash value.

representation() → str
Return the representation.

static try_parse(*string: str*) → *ValueInt16* | None
Return an int16 or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueInt32

class dsviper.**ValueInt32**(*initial_value=None*)
Bases: object

A class used to represent a value of type int32. Seamless with a Python int.

Or use Value.create(Type.INT32 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueInt32*
Return an int32 or raise.

copy() → *ValueInt32*
Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

encoded() → int
Return a Python int.

hash() → int
Return the hash value.

representation() → str
Return the representation.

static try_parse(*string: str*) → *ValueInt32* | None
Return an int32 or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueInt64

class dsviper.**ValueInt64**(*initial_value=None*)

Bases: object

A class used to represent a value of type int64. Seamless with a Python int.

Or use Value.create(Type.INT64 [, initial_value]).

ONE = 1

ZERO = 0

static cast(*value:* Value) → *ValueInt64*

Return an int64 or raise.

copy() → *ValueInt64*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → int

Return a Python int.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueInt64* | None

Return an int64 or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueFloat

class dsviper.**ValueFloat**(*initial_value=None*)

Bases: object

A class used to represent a value of type float. Seamless with a Python float.

Or use Value.create(Type.FLOAT [, initial_value]).

INF = inf

NAN = nan

NEG_INF = -inf

ONE = 1.0

ZERO = 0.0

static cast(*value:* Value) → *ValueFloat*

Return a float or raise.

copy() → *ValueFloat*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → float

Return a Python float.

hash() → int

Return the hash value.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueFloat* | None

Return a float or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueDouble

class dsviper.**ValueDouble**(*initial_value=None*)

Bases: object

A class used to represent a value of type double. Seamless with a Python float.

Or use Value.create(Type.DOUBLE [, initial_value])

INF = inf

NAN = nan

NEG_INF = -inf

ONE = 1.0

ZERO = 0.0

static cast(*value:* Value) → *ValueDouble*

Return a double or raise.

copy() → *ValueDouble*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → float

Return a Python float.

hash() → int

Return the hash value.

representation() → str
Return the representation.

static try_parse(*string: str*) → *ValueDouble* | None
Return a double or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueString

class dsviper.**ValueString**(*initial_value=None*)
Bases: object

A class used to represent a value of type string. Seamless with a Python str.

Or use Value.create(Type.STRING [, initial_value]).

EMPTY = ''

static cast(*value:* Value) → *ValueString*
Return a string or raise.

copy() → *ValueString*
Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

encoded() → str
Return a Python str.

hash() → int
Return the hash value.

representation() → str
Return the representation.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueBlob

class dsviper.**ValueBlob**(*initial_value=None*)
Bases: object

A class used to represent a value of type blob. Seamless with bytes, bytearray and base64_string.

Or use Value.create(Type.BLOB [, initial_value]).

static base64_decode(*base64_string: str*) → *ValueBlob*
Return a blob from a base64 encoded string.

base64_encode() → str

Return the base64 encoded string representation of this blob.

static cast(*value:* Value) → *ValueBlob*

Return a blob or raise.

copy() → *ValueBlob*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

embed(*name: str*) → str

Return the c++ representation of a blob.

encoded() → bytes

Return a Python bytes.

hash() → int

Return the hash value.

representation() → str

Return the representation.

sha1() → str

Return the hexdigest (SHA1).

size() → int

Return the size.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

ValueBlobId

class dsviper.**ValueBlobId**(*blob_layout*, *blob*)

Bases: object

A class used to represent a value of type blob_id. The uniq identifier is computed from the immutable content of the blob and the layout.

INVALID = 00000000000000000000000000000000000000000000

static cast(*value:* Value) → *ValueBlobId*

Return a blob_id or raise.

copy() → *ValueBlobId*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → str

Return a blob_id.

hash() → int
Return the hash value.

is_valid() → bool
Return True if the value is not INVALID.

representation() → str
Return the representation.

static try_parse(*string: str*) → *ValueBlobId* | None
Return a blob_id or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueCommitId

class dsviper.**ValueCommitId**(*initial_value=None*)
Bases: object

A class used to represent a value of type commit_id.

Use the static factory method try_parse(…).

Or use Value.create(Type.COMMIT_ID [, initial_value]).

INVALID = 0000000000000000000000000000000000000000

static cast(*value:* Value) → *ValueCommitId*
Return a commit_id or raise.

copy() → *ValueCommitId*
Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

encoded() → str
Return a Python str.

hash() → int
Return the hash value.

is_valid() → bool
Return True if the CommitId is not INVALID.

representation() → str
Return the representation.

static try_parse(*string: str*) → *ValueCommitId* | None
Return a commit_id or None.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

ValueUUId

class dsviper.**ValueUUId**(*initial_value=None*)

Bases: object

A class used to represent a value of type uuid.

Use the static factory method create(…) or try_parse(…).

Or use Value.create(Type.UUID [, initial_value]).

INVALID = 00000000-0000-0000-0000-000000000000

static cast(*value:* Value) → *ValueUUId*

Return an uuid or raise.

copy() → *ValueUUId*

Return a deep copy.

static create(*uuid_string: str* | ValueUUId | *None* = *None*) → *ValueUUId*

Return an uuid from uuid_string if specified otherwise a generated one.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

encoded() → str

Return a Python str.

hash() → int

Return the hash value.

is_valid() → bool

Return True if the value is not INVALID.

representation() → str

Return the representation.

static try_parse(*string: str*) → *ValueUUId* | None

Return an uuid or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

Container Values

dsviper.ValueVector	A class used to represent a value of type vector<element_type>.
dsviper.ValueVectorIter	Iterator for ValueVector elements.
dsviper.ValueSet	A class used to represent a value of type set<element_type>.
dsviper.ValueSetIter	Iterator for ValueSet elements.
dsviper.ValueMap	A class used to represent a value of type map<key_type, element_type>.
dsviper.ValueMapKeysIter	Iterator for ValueMap keys.

continues on next page

Table 9 – continued from previous page

dsviper.ValueMapValuesIter	Iterator for ValueMap values.
dsviper.ValueMapItemsIter	Iterator for ValueMap items (key-value pairs).
dsviper.ValueXArray	A class used to represent a value of type xarray<element_type>.
dsviper.ValueOptional	A class used to represent a value of type optional<element_type>.

ValueVector

class dsviper.**ValueVector**

Bases: object

A class used to represent a value of type vector<element_type>. Seamless with a Python seq[object].

Or use Value.create(type_vector [, initial_value]).

append(*value: _InputValues*) → None

Append value to the end of the vector.

at(*index: int*, *, *encoded: bool = True*) → _OutputValues

Return the value at index.

back(*, *encoded: bool = True*) → _OutputValues

Return the back element or raise.

capacity() → int

Return the capacity.

static cast(*value:* Value) → *ValueVector*

Return a vector<element_type> or raise.

clear() → None

Remove all items.

contains(*value: _InputValues*) → bool

Return True if the value is present.

copy() → *ValueVector*

Return a deep copy.

count(*value: _InputValues*) → int

Return the number of occurrences for the value.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

empty() → bool

Return True if the container is empty.

exchange(*idx1: int*, *idx2: int*) → None

Exchange the value at idx1 with the value at idx2.

extend(*values: Sequence* | ValueVector) → None

Extend the vector by appending the values.

front(*, *encoded: bool = True*) → _OutputValues

Return the front element or raise.

hash() → int

Return the hash value.

index(*value: _InputValues*) → int

Return the first index of value.

insert(*index: int*, *value: _InputValues*) → None

Insert object before index.

is_pack_sized() → bool

Return True if the vector packs elements in a contiguous array of binary encoded elements.

pop(*index: int = -1*, *, *encoded: bool = True*) → _OutputValues

Remove and return item at index (default last). Raises error if the vector is empty or the index is out of range.

remove(*value: _InputValues*) → None

Remove first occurrence of value. Raises error if the value is not present.

representation() → str

Return the representation.

reserve(*size: int*) → None

Reserve space for size.

resize(*size: int*) → None

Resize the vector for size.

set(*index: int*, *value: _InputValues*) → None

Set the value at index.

shrink_to_fit() → None

Shrink to fit.

size() → int

Return the number of elements.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_vector() → *TypeVector*

Return the type vector<element_type>.

ValueVectorIter

class dsviper.**ValueVectorIter**

Bases: object

Iterator for ValueVector elements.

Note: Not directly instantiable.

ValueSet

class dsviper.**ValueSet**

Bases: object

A class used to represent a value of type set<element_type>. Seamless with a Python seq[object].

Or use Value.create(type_set [, initial_value]).

add(*value: _InputValues*) → None

Add an element to a set. This has no effect if the element is already present.

at(*index: int*, *, *encoded: bool = True*) → _OutputValues

Return the element at index.

static cast(*value:* Value) → *ValueSet*

Return a set or raise.

clear() → None

Remove all elements from this set.

copy() → *ValueSet*

Return a deep copy.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

difference(*value: Sequence* | ValueSet) → *ValueSet*

Return the difference of two or more sets as a new set. (i.e., all elements that are in this set but not the others.)

difference_update(*value: Sequence* | ValueSet) → None

Remove all elements of another set from this set.

discard(*value: _InputValues*) → None

Remove an element from a set; it must be a member. If the element is not a member, do nothing.

empty() → bool

Remove True if the container is empty.

extend(*elements: Sequence* | ValueSet) → None

Add the elements of value.

hash() → int

Return the hash value.

index(*value: _InputValues*) → int | None

Return the index or None.

intersection(*value: Sequence* | ValueSet) → *ValueSet*

Return the intersection of two sets as a new set. (i.e., all elements that are in both sets.)

intersection_update(*value: Sequence* | ValueSet) → None

Update a set with the intersection of itself and another.

isdisjoint(*value: Sequence* | ValueSet) → bool

Return True if two sets have a null intersection.

issubset(*value: Sequence* | ValueSet) → bool

Report whether another set contains this set.

issuperset(*value: Sequence* | ValueSet) → bool

Report whether this set contains another set.

max(*, *encoded: bool* = *True*) → _OutputValues

Return the maximum element or raise.

min(*, *encoded: bool* = *True*) → _OutputValues

Return the minimum element or raise.

pop(*, *encoded: bool* = *True*) → _OutputValues

Remove and return the min element. Raises error if the set is empty.

pop_max(*, *encoded: bool* = *True*) → _OutputValues

Remove and return the max element. Raises error if the set is empty.

remove(*value: _InputValues*) → None

Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.

representation() → str

Return the representation.

size() → int

Return the number of elements.

symmetric_difference(*value: Sequence* | ValueSet) → *ValueSet*

Return the symmetric difference of two sets as a new set. (i.e., all elements that are in exactly one of the sets.)

symmetric_difference_update(*value: Sequence* | ValueSet) → None

Update a set with the symmetric difference of itself and another.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_set() → *TypeSet*

Return the type set<element_type>.

union(*value: Sequence* | ValueSet) → *ValueSet*

Return the union of sets as a new set. (i.e., all elements that are in either set).

update(*value: Sequence* | ValueSet) → None

Update a set with the union of itself and other.

ValueSetIter

class dsviper.**ValueSetIter**

Bases: object

Iterator for ValueSet elements.

Note: Not directly instantiable.

ValueMap

class dsviper.**ValueMap**

Bases: object

A class used to represent a value of type map<key_type, element_type>. Seamless with a Python dict[object, object].

Or use Value.create(type_map [, initial_value]).

at(*key: _InputValues*, *, *encoded: bool = True*) → _OutputValues

Return the value for a key.

static cast(*value:* Value) → *ValueMap*

Return a map or raise.

clear() → None

Remove all items.

contains(*key: _InputValues*) → bool

Return True if the value is present.

copy() → *ValueMap*

Return a deep copy.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

discard(*key: _InputValues*) → None

Remove the key if present.

empty() → bool

Return True if the container is empty.

get(*key: _InputValues*, *default: _InputValues | None = None*) → None | _OutputValues

Return the value associated with the key or default.

hash() → int

Return the hash value.

items(*, *encoded: bool = True*) → *ValueMapItemsIter*

Return a MapItemsIter.

keys(*, *encoded: bool = True*) → *ValueMapKeysIter*

Return a MapKeysIter.

max(*, *encoded: bool = True*) → _OutputValues

Return the max key.

min(*, *encoded: bool = True*) → _OutputValues

Return the min key.

pop(*key: _InputValues*, *default: _InputValues | None = None*, *, *encoded: bool = True*) → _OutputValues

Remove the key and return the corresponding value. If the key is not found, return the default if given; otherwise raise.

popitem() → *ValueTuple*

Remove and return a (key, value) pair as a 2-tuple. Raises error if the map is empty.

remove(*key: _InputValues*) → None

Remove the key or raises.

representation() → str

Return the representation.

set(*key: _InputValues*, *value: _InputValues*) → None

Set the value for a key.

setdefault(*key: _InputValues*, *value: _InputValues*) → None

Insert the value for the key if the key is not in the map. Return the value for the key if the key is in the map, else default.

size() → int

Return the number of elements.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_map() → *TypeMap*

Return the type map<key_type, element_type>.

update(*value: _InputValues*) → None

for k, v in other: self[key] = v.

values(*, *encoded: bool = True*) → *ValueMapValuesIter*

Return a MapValuesIter.

ValueMapKeysIter

class dsviper.**ValueMapKeysIter**

Bases: object

Iterator for ValueMap keys.

Note: Not directly instantiable.

ValueMapValuesIter

class dsviper.**ValueMapValuesIter**

Bases: object

Iterator for ValueMap values.

Note: Not directly instantiable.

ValueMapItemsIter

class dsviper.**ValueMapItemsIter**

Bases: object

Iterator for ValueMap items (key-value pairs).

Note: Not directly instantiable.

ValueXArray

class dsviper.**ValueXArray**

Bases: object

A class used to represent a value of type xarray<element_type>. Seamless with a Python seq[object].

Or use Value.create(type_xarray [, initial_value]).

END = 00000000-0000-0000-0000-000000000000

append(*value: _InputValues*) → *ValueUUId*

Append value to the end.

at(*position:* ValueUUId, *, encoded: bool = True*) → _OutputValues | None

Return the element at the position or None.

static cast(*value:* Value) → *ValueXArray*

Return a xarray<element_type> or raise.

copy() → *ValueXArray*

Return a deep copy.

static create_position() → *ValueUUId*

Create and return a position.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

disable_position(*position:* ValueUUId) → None

Disable the position.

extend(*values: Sequence* | ValueVector) → *ValueUUId*

Extend by appending the values.

has_position(*position:* ValueUUId) → bool

Return True if the position exists.

hash() → int

Return the hash value.

index(*position:* ValueUUId) → int | None

Return the index of the position.

insert(*before_position:* ValueUUId, *value: _InputValues*, *new_position:* ValueUUId | *None = None*) → *ValueUUId*

Insert the value before_position and return the new position. Use new_position if specified, else a new position is created.

insert_position(*before_position:* ValueUUId, *new_position:* ValueUUId) → None

Insert before_position a new_position.

items() → list[tuple[*ValueUUId*, *Value*]]

Return the list of positions with a value.

position(*index: int*) → *ValueUUId* | None

Return the position for the index or None.

position_of(*value: _InputValues*) → *ValueUUId* | None
Return the position of the value or None.

positions() → list[*ValueUUId*]
Return the list of positions.

remove(*position:* ValueUUId) → None
Remove the element at the position.

representation() → str
Return the representation.

set(*position:* ValueUUId, *value: _InputValues*)
Set the value at the position.

to_vector() → *ValueVector*
Convert to a vector<element_type>.

type() → *Type*
Return the type.

type_code() → str
Return the type code.

type_xarray() → *TypeXArray*
Return the type xarray<element_type>.

ValueOptional

class dsviper.**ValueOptional**(*type_optional*, *initial_value*)
Bases: object

A class used to represent a value of type optional<element_type>.

Or use Value.create(type_optional [, initial_value]).

static cast(*value:* Value) → *ValueOptional*
Return an optional or raise.

clear() → None
Clear the container.

copy() → *ValueOptional*
Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str
Return the description.

get(*default: _InputValues* | *None* = *None*, *, *encoded: bool* = *False*) → _OutputValues
Return the wrapped value or return default if given; otherwise raise.

is_nil() → bool
Return True if there is no value to unwrap.

representation() → str
Return the representation.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_optional() → *TypeOptional*

Return the type optional<element_type>.

unwrap(*, *encoded: bool = False*) → _OutputValues

Return the wrapped value or raise.

wrap(*value: _InputValues*) → None

Wrap the value.

Algebraic Values

dsviper.ValueTuple	A class used to represent a value of type tuple<T0, ...>.
dsviper.ValueTupleIter	Iterator for ValueTuple elements.
dsviper.ValueVec	ValueVec(type_vec [, initial_value]).
dsviper.ValueMat	A class used to represent a value of type mat<element_type, columns, rows>.
dsviper.ValueVariant	ValueVariant(type_variant, initial_value).
dsviper.ValueAny	A class used to represent a value of any type.

ValueTuple

class dsviper.**ValueTuple**

Bases: object

A class used to represent a value of type tuple<T0, …>. Seamless with a Python seq[object].

Or use Value.create(type_tuple [, initial_value]).

at(*index: int*, *, *encoded: bool = True*) → _OutputValues

Return the value at index.

static cast(*value:* Value) → *ValueTuple*

Return a tuple or raise.

copy() → *ValueTuple*

Return a deep copy.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

empty() → bool

Remove True if the container is empty.

hash() → int

Return the hash value.

replace(*index: int*, *value: _InputValues*) → *ValueTuple*

Return a copy of the tuple where the value was replaced.

representation() → str

Return the representation.

set(*index: int*, *value: _InputValues*) → None

Set the value at index.

size() → int

Return the number of elements.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_tuple() → *TypeTuple*

Return the type tuple<T0,…>.

ValueTupleIter

class dsviper.**ValueTupleIter**

Bases: object

Iterator for ValueTuple elements.

Note: Not directly instantiable.

ValueVec

class dsviper.**ValueVec**

Bases: object

ValueVec(type_vec [, initial_value]). A class used to represent a value of type vec<element_type, size>. Seamless with a Python seq[int|real].

Or use Value.create(type_vec [, initial_value]).

at(*index: int*, *, *encoded: bool = True*) → int | float

Return the value at index.

static cast(*value:* Value) → *ValueVec*

Return a vec<element_type, size> or raise.

copy() → *ValueVec*

Return a deep copy.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

hash() → int

Return the hash value.

representation() → str

Return the representation.

set(*index: int*, *value: _NumericInputValues*) → None

Set the value at index.

size() → int

Return the number of elements.

to_tuple() → tuple

Return a tuple.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_vec() → *TypeVec*

Return the type vec<element_type, size>.

ValueMat

class dsviper.**ValueMat**

Bases: object

A class used to represent a value of type mat<element_type, columns, rows>. Seamless with a Python seq[seq[int|float]].

Or use Value.create(type_mat [, initial_value]).

at(*column: int*, *row: int*, *, *encoded: bool = True*) → _OutputValues

Return the value at column and row.

static cast(*value:* Value) → *ValueMat*

Return a mat or raise.

columns() → int

Return the number of columns.

copy() → *ValueMat*

Return a deep copy.

description(*, *namespace:* NameSpace | *None = None*) → str

Return the description.

hash() → int

Return the hash value.

representation() → str

Return the representation.

rows() → int

Return the number of rows.

set(*column: int*, *row: int*, *value: _InputValues*) → None

Set the value at column, row.

size() → int

Return the number of elements.

to_tuple() → tuple

Return a tuple.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_mat() → *TypeMat*

Return the type mat<element_type, size>.

ValueVariant

class dsviper.**ValueVariant**

Bases: object

ValueVariant(type_variant, initial_value). A class used to represent a value of type variant<T0, …>. Seamless with a Python object compatible with the variant's types.

Or use Value.create(type_variant [, initial_value]).

static cast(*value:* Value) → *ValueVariant*

Return a variant<T0, …> or raise.

copy() → *ValueVariant*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

hash() → int

Return the hash value.

representation() → str

Return the representation.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_variant() → *TypeVariant*

Return the type variant<T0, …>.

unwrap(*, *encoded: bool* = *True*) → _OutputValues

Return the value or raise.

wrap(*value: _InputValues*, *type:* Type | *None* = *None*) → None

wrap a value.

ValueAny

class dsviper.**ValueAny**(*initial_value=None*)

Bases: object

A class used to represent a value of any type. Seamless with many Python objects.

Or use Value.create(Type.ANY [, initial_value]).

static cast(*value:* Value) → *ValueAny*

Return an any or raise.

clear() → None

Clear the container.

copy() → *ValueAny*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

hash() → int

Return the hash value.

is_nil() → bool

Return True if there is no value to unwrap.

representation() → str

Return the representation.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

unwrap(*, *encoded: bool* = *True*) → _OutputValues

Return the wrapped value or raise.

wrap(*value: _InputValues*) → None

Wrap a value.

User-Defined Values

dsviper.ValueStructure	A class used to represent a value of type struct.
dsviper.ValueEnumeration	A class used to represent a value of type struct.
dsviper.ValueKey	A class used to represent a value of type key<element_type>.

ValueStructure

class dsviper.**ValueStructure**

Bases: object

A class used to represent a value of type struct. Seamless with a Python dict[str, object].

Or use Value.create(type_structure [, initial_value]).

at(*field_name*, *, *encoded: bool* = *True*) → _OutputValues

Return the value of the field.

static cast(*value:* Value) → *ValueStructure*

Return a struct or raise.

copy() → *ValueStructure*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

hash() → int

Return the hash value.

representation() → str

Return the representation.

set(*field_name: str*, *value: _InputValues*)

Set the value of the field.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_structure() → *TypeStructure*

Return the type struct.

ValueEnumeration

class dsviper.**ValueEnumeration**

Bases: object

A class used to represent a value of type struct. Seamless with a Python str.

Or use Value.create(type_enumeration [, initial_value]). If the initial_value is not specified, the first enum case is used.

The string representation of an enum case start with a '.'. ex: Value.create(type_enum_E, '.a') or Value.create(type_enum_E, 'E.a').

static cast(*value:* Value) → *ValueEnumeration*

Return an enum or raise.

copy() → *ValueEnumeration*

Return a deep copy.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

index() → int

Return the index.

name() → str

Return the name of the enumeration case.

representation() → str

Return the representation.

static try_parse(*string: str*, *type_enumeration:* TypeEnumeration) → *ValueEnumeration* | None

Return an enum or None.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_enumeration() → *TypeEnumeration*

Return the type enum.

ValueKey

class dsviper.**ValueKey**

Bases: object

A class used to represent a value of type key<element_type>. Use the static factory method create(…).

Note: Not directly instantiable.

static cast(*value:* Value) → *ValueKey*

Return a key or raise.

copy() → *ValueKey*

Return a deep copy.

static create(*type_concept:* TypeConcept, *instance_id: str* | ValueUUId | *None* = *None*) → *ValueKey*

Return a new key with the instance_id or a generated one if not specified.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

detail_representation() → str

Return a detailed representation.

detail_type_representation() → str

Return the detail type representation.

has_parent_key() → bool

Return True if a parent key is available.

hash() → int

Return the hash value.

instance_id() → *ValueUUId*

Return the uuid of the instance.

is_member(*target_concept:* TypeConcept) → bool

Return True if target_concept is a member.

static keys(*key:* ValueKey) → list[*ValueKey*]

Return the list of hierarchical keys.

representation() → str

Return the representation.

to_any_concept_key() → *ValueKey*

Return an key<any_concept>.

to_club_key(*type_club:* TypeClub) → *ValueKey*

Return a key<type_club> or raise.

to_concept_key() → *ValueKey*

Return a key<type_concept>.

to_key(*type_key:* TypeKey) → *ValueKey*

Return a key<type_key> or raise.

to_member_key(*target_concept:* TypeConcept) → *ValueKey*

Return a key<target_concept> or raise.

to_parent_key() → *ValueKey*

Return a key<parent_concept> or raise.

type() → *Type*

Return the type.

type_code() → str

Return the type code.

type_concept() → *TypeConcept*

Return the type of the concept.

type_key() → *TypeKey*

Return the TypeKey.

Value Program

A `ValueProgram` is a sequence of opcodes that describe mutations to values. Each opcode represents an atomic operation (set, update, union, subtract, etc.).

dsviper.ValueProgram	A class used to retreive opcodes.
dsviper.ValueOpcodeKey	A class used to retreive the mutation opcodes for an instance of a concept for an attachment.
dsviper.ValueOpcode	A utility class to handle opcodes encoder and streamer.
dsviper.ValueOpcodeDocumentSet	A class used to represent the type and the arguments of an opcode.
dsviper.ValueOpcodeDocumentUpdate	A class used to represent the type and the arguments of a opcode.
dsviper.ValueOpcodeMapUnion	A class used to represent the type and the arguments of a opcode.
dsviper.ValueOpcodeMapSubtract	A class used to represent the type and the arguments of a opcode.
dsviper.ValueOpcodeMapUpdate	A class used to represent the type and the arguments of a opcode.
dsviper.ValueOpcodeSetUnion	A class used to represent the type and the arguments of an opcode.
dsviper.ValueOpcodeSetSubtract	A class used to represent the type and the arguments of an opcode.
dsviper.ValueOpcodeXArrayInsert	A class used to represent the type and the arguments of an opcode.
dsviper.ValueOpcodeXArrayRemove	A class used to represent the type and the arguments of an opcode.

continues on next page

Table 12 – continued from previous page

dsviper.ValueOpcodeXArrayUpdate	A class used to represent the type and the arguments of an opcode.

ValueProgram

class dsviper.**ValueProgram**

Bases: object

A class used to retreive opcodes.

Note: Not directly instantiable.

all_opcodes() → list[*ValueOpcode*]

Return the list of all opcodes.

document_set_keys(*attachment_runtime_id:* ValueUUId) → set[*ValueOpcodeKey*]

Return the set of ValueOpcodeKey of type 'Document_Set' associated with the attachment_runtime_id.

has_document_set(*opcode_key:* ValueOpcodeKey) → bool

Return True if an opcode 'Document_Set' is associated with the opcode_key.

opcodes(*attachment_runtime_id:* ValueUUId, *key:* ValueKey) → list[*ValueOpcode*]

Return the list of opcodes associated with attachment_runtime_id and the key.

ValueOpcodeKey

class dsviper.**ValueOpcodeKey**

Bases: object

A class used to retreive the mutation opcodes for an instance of a concept for an attachment. Only used by the evaluator.

Note: Not directly instantiable.

attachment_runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

concept_runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime for the concept's type.

instance_id() → *ValueUUId*

Return the uuid of the instance of the concept.

ValueOpcode

class dsviper.**ValueOpcode**

Bases: object

A utility class to handle opcodes encoder and streamer.

Note: Not directly instantiable.

static decode(*blob:* ValueBlob, *definitions:* DefinitionsConst, *, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → list[*ValueOpcode*]

Return a list of opcodes by decoding the blob with a StreamBinaryCodec if not specified. .

static encode(*opcodes: list[*ValueOpcode*]*, *, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *ValueBlob*

Return a blob that encodes the list of opcodes with a StreamBinaryCodec if not specified.

static read(*stream_reading:* StreamReading, *definitions:* DefinitionsConst) → list[*ValueOpcode*]

Read and return the list of opcodes from the stream.

static write(*opcodes: list[*ValueOpcode*]*, *stream_writing:* StreamWriting) → None

Write the list of opcodes to the stream.

ValueOpcodeDocumentSet

class dsviper.**ValueOpcodeDocumentSet**

Bases: object

A class used to represent the type and the arguments of an opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *Value*]

Return the tuple(attachment, key, value).

key() → *ValueOpcodeKey*

Return the key.

type() → str

Return the type.

value() → *Value*

Return the value.

ValueOpcodeDocumentUpdate

class dsviper.**ValueOpcodeDocumentUpdate**

Bases: object

A class used to represent the type and the arguments of a opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *Value*]

Return the tuple(attachment, key, path, value).

key() → *ValueOpcodeKey*

Return the key.

path() → *PathConst*

Return the path.

type() → str

Return the type.

value() → *Value*

Return the value.

ValueOpcodeMapUnion

class dsviper.**ValueOpcodeMapUnion**

Bases: object

A class used to represent the type and the arguments of a opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueMap*]

Return the tuple(attachment, key, path, value).

key() → *ValueOpcodeKey*

Return the key.

path() → *PathConst*

Return the path.

type() → str

Return the type.

value() → *ValueMap*

Return the value.

ValueOpcodeMapSubtract

class dsviper.**ValueOpcodeMapSubtract**

Bases: object

A class used to represent the type and the arguments of a opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueSet*]

Return the tuple(attachment, key, path, value).

key() → *ValueOpcodeKey*

Return the key.

path() → *PathConst*

Return the path.

type() → str

Return the type.

value() → *ValueSet*

Return the value.

ValueOpcodeMapUpdate

class dsviper.**ValueOpcodeMapUpdate**

Bases: object

A class used to represent the type and the arguments of a opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueMap*]

Return the tuple(attachment, key, path, value).

key() → *ValueOpcodeKey*
Return the key.

path() → *PathConst*
Return the path.

type() → str
Return the type.

value() → *ValueMap*
Return the value.

ValueOpcodeSetUnion

class dsviper.**ValueOpcodeSetUnion**
Bases: object

A class used to represent the type and the arguments of an opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueSet*]
Return the tuple(attachment, key, path, value).

key() → *ValueOpcodeKey*
Return the key.

path() → *PathConst*
Return the path.

type() → str
Return the type.

value() → *ValueSet*
Return the value.

ValueOpcodeSetSubtract

class dsviper.**ValueOpcodeSetSubtract**
Bases: object

A class used to represent the type and the arguments of an opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueSet*]
Return the tuple(attachment, key, path, value).

key() → *ValueOpcodeKey*
Return the key.

path() → *PathConst*
Return the path.

type() → str
Return the type.

value() → *ValueSet*
Return the value.

ValueOpcodeXArrayInsert

class dsviper.**ValueOpcodeXArrayInsert**

Bases: object

A class used to represent the type and the arguments of an opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueUUId*, *ValueUUId*]

Return the tuple(attachment, key, value, before_position, new_position).

before_position() → *ValueUUId*

Return the before position.

key() → *ValueOpcodeKey*

Return the key.

path() → *PathConst*

Return the path.

position() → *ValueUUId*

Return the position.

type() → str

Return the type.

ValueOpcodeXArrayRemove

class dsviper.**ValueOpcodeXArrayRemove**

Bases: object

A class used to represent the type and the arguments of an opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → Tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueUUId*]

Return the tuple(attachment, key, value, position).

key() → *ValueOpcodeKey*

Return the key.

path() → *PathConst*

Return the path.

position() → *ValueUUId*

Return the position.

type() → str

Return the type.

ValueOpcodeXArrayUpdate

class dsviper.**ValueOpcodeXArrayUpdate**

Bases: object

A class used to represent the type and the arguments of an opcode.

Note: Not directly instantiable.

arguments(*definitions:* DefinitionsConst) → tuple[*Attachment*, *ValueKey*, *PathConst*, *ValueUUId*, *Value*]

Return the tuple(attachment, key, path, position, value).

key() → *ValueOpcodeKey*

Return the key.

path() → *PathConst*

Return the path.

position() → *ValueUUId*

Return the position.

type() → str

Return the type.

value() → *Value*

Return the value.

3.3.3 Attachments

Attachments connect values to documents via keys. They are the fundamental mechanism for associating typed data with document instances.

When to use: Use attachments to store and retrieve values associated with document keys. The `AttachmentGetting` interface provides read access, while `AttachmentMutating` extends it with write operations.

Quick Start

```
from dsviper import CommitDatabase, CommitMutableState

db = CommitDatabase.open("model.cdb")
db.definitions().inject()

# Read via AttachmentGetting
state = db.state(db.last_commit_id())
getting = state.attachment_getting()
value = getting.get(MYAPP_A_USER, user_key)

# Write via AttachmentMutating
mutable = CommitMutableState(state)
mutating = mutable.attachment_mutating()
mutating.set(MYAPP_A_USER, user_key, document)
```

Core Classes

dsviper.Attachment	A class used to represent an attachment.

Attachment

class dsviper.**Attachment**

Bases: `object`

A class used to represent an attachment.

Note: Not directly instantiable.

create_document() → *Value*

Return a new document.

create_key(*instance_id:* ValueUUId | *None* = *None*) → *ValueKey*

Return a new key.

create_structure(*initial_value: dict[str, _InputValues]* | *None* = *None*) → *ValueStructure*

Create a new structure and may initialize it or raise.

description(*, *namespace:* NameSpace | *None* = *None*) → str

Return the description.

document_type() → *Type*

Return the type of the document.

documentation() → str

Return the documentation.

identifier() → str

Return the identifier.

key_type() → _AbstractionTypes

Return the type of the key.

key_type_name() → *TypeName*

Return the key's TypeName.

keys_type() → *TypeSet*

Return the type of the key.

optional_document_type() → *TypeOptional*

Return the type of the optional document.

representation(*, *namespace:* NameSpace | *None* = *None*) → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_key() → *TypeKey*

Return the type key<element_type>.

type_name() → *TypeName*

Return the TypeName.

Attachment Interfaces

These interfaces provide read (getting) and write (mutating) access to attached values.

dsviper.AttachmentGetting	An interface used to retreive a value (aka document) state.
dsviper.AttachmentMutating	An interface used to express fine-grained mutations of a value.

AttachmentGetting

class dsviper.**AttachmentGetting**

Bases: object

An interface used to retreive a value (aka document) state.

Note: Not directly instantiable.

attachment(*attachment_runtime_id:* ValueUUId) → *Attachment*

Return the attachment or raise.

definitions() → *DefinitionsConst*

Return the definitions.

static diff_keys(*current_attachment_getting:* AttachmentGetting, *other_attachment_getting:* AttachmentGetting, *attachment:* Attachment) → tuple[*ValueSet*, *ValueSet*, *ValueSet*, *ValueSet*]

Return the tuple (added_keys, removed_keys, different_keys, same_keys).

- added_keys: keys present in 'other' and not in 'current'.
- removed_keys: keys present in 'current' and not in 'other'.
- different_keys: documents present in both but not equal.
- same_keys: documents present in both and equal.

enumerate(*attachment:* Attachment, *, *encoded: bool = True*) → list[tuple[*ValueKey*, _OutputValues]]

Return a list of (key, document).

get(*attachment:* Attachment, *key:* ValueKey) → *ValueOptional*

Return an optional<document_type> associated with the key in the attachment.

has(*attachment:* Attachment, *key:* ValueKey) → bool

Return True if a document is present for the key.

keys(*attachment:* Attachment) → *ValueSet*

Return all keys.

AttachmentMutating

class dsviper.**AttachmentMutating**

Bases: object

An interface used to express fine-grained mutations of a value.

Note: Not directly instantiable.

attachment(*attachment_runtime_id:* ValueUUId) → *Attachment*

Return the attachment or raise.

definitions() → *DefinitionsConst*

Return the definition.

diff(*attachment:* Attachment, *key:* ValueKey, *value: _InputValues*, *, *recursive: bool = False*) → None

Applies the mutation opcodes from the calculation of the difference between the value passed in parameter and the current value in the state. If recursive is True, the difference is recursively computed by structure's field.

enumerate (*attachment:* Attachment, *, *encoded: bool = True*) → list[tuple[*ValueKey*, _OutputValues]]

Return the list (key, document).

get (*attachment:* Attachment, *key:* ValueKey) → *ValueOptional*

Return an optional<document_type> associated with the key.

has (*attachment:* Attachment, *key:* ValueKey) → bool

Return True if a document is associated with the key.

insert_in_xarray (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *before_position:* ValueUUId, *new_position:* ValueUUId, *value: _InputValues*) → None

Insert a value by creating a new_position before before_position.

keys (*attachment:* Attachment) → *ValueSet*

Return all keys.

remove_in_xarray (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *position:* ValueUUId) → None

Remove the position.

set (*attachment:* Attachment, *key:* ValueKey, *value: _InputValues*) → None

Associate the document with the key.

subtract_in_map (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Form the subtraction between the map located at path in the document associated with the key and the value.

subtract_in_set (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Form the subtraction between the set at path in the document associated with the key and the value.

union_in_map (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Form the union between the map located at path in the document associated with the key and the value.

union_in_set (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Form the union between the set at path in the document associated with key and the value.

update (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Update the value located at path in the document associated with the key.

update_in_map (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Form the subtraction between the map located at path in the document associated with the key and the value.

update_in_xarray (*attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *position:* ValueUUId, *value: _InputValues*) → None

Update the value at position.

Attachment Functions

User-defined functions that operate on attached values, defined via DSM.

dsviper.AttachmentGettingFunction	A class used to call a C++ function.
dsviper.AttachmentMutatingFunction	A class used to call a mutable function.
dsviper.AttachmentFunctionPool	A class used to register and retrieve C++ attachment function.
dsviper.AttachmentFunctionPoolFunctions	A class used to represent the functions of a pool.

AttachmentGettingFunction

class dsviper.**AttachmentGettingFunction**

Bases: object

A class used to call a C++ function.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

prototype() → *FunctionPrototype*

Return the prototype.

AttachmentMutatingFunction

class dsviper.**AttachmentMutatingFunction**

Bases: object

A class used to call a mutable function.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

prototype() → *FunctionPrototype*

Return the prototype.

AttachmentFunctionPool

class dsviper.**AttachmentFunctionPool**

Bases: object

A class used to register and retrieve C++ attachment function.

Note: Not directly instantiable.

check(*func_name: str*) → *AttachmentMutatingFunction* | *AttachmentGettingFunction*

Return the function or raise.

documentation() → str

Return the documentation.

funcs → *AttachmentFunctionPoolFunctions*

all functions.

name() → str

Return the name.

query(*func_name: str*) → *AttachmentMutatingFunction* | *AttachmentGettingFunction* | None

Return the function or None.

uuid() → *ValueUUId*

Return the uuid.

AttachmentFunctionPoolFunctions

class dsviper.**AttachmentFunctionPoolFunctions**

Bases: object

A class used to represent the functions of a pool.

Note: Not directly instantiable.

3.3.4 Database

Database classes provide persistence for Viper documents using SQLite or remote connections.

When to use: Use `Database` for simple key-value persistence without history tracking. For versioned data, see *Commit System*.

Quick Start

```
from dsviper import Database, DSMBuilder

# Create or open database
db = Database.create("data.vdb")

# Load definitions from DSM
builder = DSMBuilder.assemble("model.dsm")
report, dsm_defs, defs = builder.parse()
db.extend_definitions(defs)

# Write (requires transaction)
db.begin_transaction()
db.set(attachment, key, document)
db.commit()

# Read
result = db.get(attachment, key)
if not result.is_nil():
    doc = result.unwrap()

# Delete
db.begin_transaction()
db.delete(attachment, key)
db.commit()
```

See also

For detailed examples with concrete attachments, see *Database*.

Choosing the Right Class

Use Case	Class	Note
Simple CRUD, no history	`Database`	Creates `.vdb` files
Need version history	`CommitDatabase`	See *Commit System*
Remote database access	`DatabaseRemote`	Client-server mode

Core Classes

dsviper.Database	A Database is a CRUD like transactional database.
dsviper.DatabaseSQLite	A low-level class used to represent a CRUD like database based on SQLite3 through the Databasing interface.
dsviper.DatabaseRemote	A low-level class used to access a CRUD like database from a remote repository through the Databasing interface.
dsviper.Databasing	An interface used to abstract the implementation of the persistence layer for a CRUD like database.

Database

class dsviper.**Database**

Bases: `object`

A Database is a CRUD like transactional database. Use the static factory method create_in_memory(), create(…), open(…), connect(…) or connect_local(…).

Note: Not directly instantiable.

attachment_getting() → *AttachmentGetting*

Return the AttachmentGetting interface.

begin_transaction(*mode: str | None = None*) → None

Begin a transaction where the mode is 'Deferred', 'Immediate' or 'Exclusive'.

blob(*blob_id:* ValueBlobId) → *ValueBlob*

Return a blob or None.

blob_getting() → *BlobGetting*

Return the BlobGetting interface.

blob_ids() → set[*ValueBlobId*]

Return the set of blob_id.

blob_info(*blob_id:* ValueBlobId) → *BlobInfo*

Return the information for the blob or None.

blob_infos(*blob_ids: set[*ValueBlobId*]*) → list[*BlobInfo*]

Return a list of BlobInfo.

blob_statistics() → *BlobStatistics*

Return the statistics for blobs.

blob_stream_append(*blob_stream:* BlobStream, *blob:* ValueBlob) → None

Append a blob to the stream.

blob_stream_close(*blob_stream:* BlobStream) → *ValueBlobId*

Return the computed blob_id and close the stream.

blob_stream_create(*blob_layout:* BlobLayout, *size: int*) → *BlobStream*

Return a stream to fill a blob.

close() → None

Close the database.

codec_name() → str

Return the name of the codec.

commit()

Commit the transaction.

static connect(*database: str*, *host: str*, *service: str = '54322'*) → *Database*

Connect to a server and use the database.

static connect_local(*database: str*, *socket_path: str*) → *Database*

Connect to socket_path and use the database.

static create(*file_path: str*, *, *documentation: str | None = None*) → *Database*

Create and return a database.

create_blob(*blob_layout:* BlobLayout, *blob:* ValueBlob) → *ValueBlobId*

Create the blob and return the blob_id.

create_blob_from_buffer(*buffer*) → *ValueBlobId*

create_blob_buffer($self, buffer) –

Compute and return the blob_id for a object implementing protocol buffer.

static create_in_memory() → *Database*

Create and return a database in memory.

static databases(*host: str*, *service: str = '54322'*) → list[str]

Connect to a server and return the list of databases.

static databases_local(*socket_path: str*) → list[str]

Connect to a server and return the list of databases.

databasing() → *Databasing*

Return the Databasing interface.

definitions() → *DefinitionsConst*

Return the definitions.

definitions_hexdigest() → str

Return the hexdigest of the definitions.

del_blob(*blob_id:* ValueBlobId) → bool

Return True if the blob was deleted.

delete(*attachment:* Attachment, *key:* ValueKey) → bool

Return True if the value was deleted.

documentation() → str

Return the documentation.

extend_definitions(*definitions:* DefinitionsConst) → *DefinitionsExtendInfo*

Extend the definitions.

get(*attachment:* Attachment, *key:* ValueKey) → *ValueOptional*

Return and optional<document_type> associated with the key.

has(*attachment:* Attachment, *key:* ValueKey) → bool

Return True if a document is present for the key.

in_memory() → bool

Return True if the database is in memory.

in_transaction() → bool

Return True if a transaction is running.

is_closed() → bool

Return True if the database is closed.

static is_compatible(*file_path: str*) → bool

Return True if the SQL schema contains required tables.

keys(*attachment:* Attachment) → *ValueSet*

Return the set of all keys.

static open(*file_path: str*, *readonly: bool = False*) → *Database*

Open and return a database.

path() → str

Return the path.

read_blob(*blob_id:* ValueBlobId, *size: int*, *offset: int*) → *ValueBlob*

Return the blob at offset.

rollback()

Roll back the transaction.

set(*attachment:* Attachment, *key:* ValueKey, *value: _InputValues*) → bool

Assigns the value to the key.

uuid() → *ValueUUId*

Return the uuid.

DatabaseSQLite

class dsviper.**DatabaseSQLite**

Bases: object

A low-level class used to represent a CRUD like database based on SQLite3 through the Databasing interface. Use the high-level static factory method Database.create_in_memory(), Database.create(…) or Database.open(…).

Note: Not directly instantiable.

close()

Close the database.

static create(*file_path: str*, *, *documentation: str | None = None*) → *DatabaseSQLite*

Create and return a database.

static create_in_memory() → *DatabaseSQLite*

Create and return a database in memory.

databasing() → *Databasing*

Return the Databasing interface.

is_closed() → bool

Return True if the database is closed.

static is_compatible(*file_path: str*) → bool

Return True the SQL schema contains required tables.

static open(*file_path: str*, *readonly: bool = False*) → *DatabaseSQLite*

Open and return a database.

sqlite() → *SQLite*

Return the sqlite api.

DatabaseRemote

class dsviper.**DatabaseRemote**

Bases: object

A low-level class used to access a CRUD like database from a remote repository through the Databasing interface. Use the high-level static factory method Database.connect(…) or Database.connect_local(…). Use the high-level static method Database.databases(…) or Database.databases_local(…) to retrieve the list of the databases from a server.

Note: Not directly instantiable.

close() → None

Close the database.

static connect(*database: str*, *host: str*, *service: str = '54322'*) → *DatabaseRemote*

Connect to a server and use the database.

static connect_local(*database: str*, *socket_path: str*) → *DatabaseRemote*

Connect to socket_path and use the database.

databases(*host: str*, *service: str = '54322'*) → list[str]

Connect to a server and return the list of databases.

databases_local(*socket_path: str*) → list[str]

Connect to a server and return the list of databases.

databasing() → *Databasing*

Return the Databasing interface.

is_closed() → bool

Return True the connection is closed.

Databasing

class dsviper.**Databasing**

Bases: object

An interface used to abstract the implementation of the persistence layer for a CRUD like database.

Note: Not directly instantiable.

TRANSACTION_DEFERRED = 'Deferred'

TRANSACTION_EXCLUSIVE = 'Exclusive'

TRANSACTION_IMMEDIATE = 'Immediate'

begin_transaction(*mode: str | None = None*) → None

Begin a transaction where the mode is 'Deferred', 'Immediate' or 'Exclusive'.

blob(*blob_id:* ValueBlobId) → *ValueBlob* | None

Return a blob or None.

blob_ids() → set[*ValueBlobId*]

Return the set of blob_id of all blobs.

blob_info(*blob_id:* ValueBlobId) → *BlobInfo* | None

Return the information for the blob or None.

blob_infos(*blob_ids: set[*ValueBlobId*]*) → list[*BlobInfo*]

Return a list of BlobInfo.

blob_statistics() → *BlobStatistics*

Return the statistics for blobs.

blob_stream_close(*stream_id:* ValueUUId, *blob_id:* ValueBlobId) → None

Close the stream.

blob_stream_create(*blob_layout:* BlobLayout, *size: int*) → *ValueUUId*

Return the uuid of the stream.

blob_stream_delete(*stream_id:* ValueUUId) → None

Delete the stream.

blob_stream_write(*stream_id:* ValueUUId, *blob:* ValueBlob, *offset: int*) → None

write a region of the blob.

close() → None

Close the database.

codec_name() → str

Return the name of the codec.

commit() → None

Commit the transaction.

create_blob(*blob_layout:* BlobLayout, *blob:* ValueBlob) → *ValueBlobId*

Create the blob and return the blob_id.

create_zero_blob(*blob_id:* ValueBlobId, *blob_layout:* BlobLayout, *size: int*) → bool

Return the True if the blob was created.

data_version() → int

Return the data version.

definitions() → *DefinitionsConst*

Return the definitions.

definitions_hexdigest() → str

Return the definitions.

del_blob(*blob_id:* ValueBlobId) → bool

Return True if the blob was deleted.

delete(*attachment:* Attachment, *key:* ValueKey) → bool

Return True if the value was deleted.

documentation() → str
Return the documentation.

extend_definitions(*other:* DefinitionsConst) → *DefinitionsExtendInfo*
Extend the definitions.

freeze_blob(*blob_id:* ValueBlobId) → bool
Return True if the blob was frozen.

get(*attachment:* Attachment, *key:* ValueKey) → *ValueOptional*
Return an optional<document_type> associated with the key.

has(*attachment:* Attachment, *key:* ValueKey) → bool
Return True if a value is present for the key.

in_transaction() → bool
Return True if a transaction is running.

is_closed() → bool
Return True if the database is closed.

keys(*attachment:* Attachment) → *ValueSet*
Return the set of all keys.

path() → str
Return the path.

read_blob(*blob_id:* ValueBlobId, *size: int*, *offset: int*) → *ValueBlob*
Return the blob at offset.

rollback() → None
Roll back the transaction.

set(*attachment:* Attachment, *key:* ValueKey, *value: _InputValues*) → bool
Assigns the value to the key.

uuid() → *ValueUUId*
Return the uuid.

write_blob(*blob_id:* ValueBlobId, *blob:* ValueBlob, *offset: int*) → None
Write a region of the blob at offset.

Low-Level

dsviper.SQLite	A class used to retreive the configuration of SQLite3.

SQLite

class dsviper.**SQLite**
Bases: object

A class used to retreive the configuration of SQLite3.

Note: Not directly instantiable.

compile_options() → dict[str, str]

Return a dict[str, str] of compilation options

get_pragma(*key*) → list[str]

Return the pragma value in a list of str.

static is_sqlite3(*file_path: str*) → bool

Return True if the file is a SQLite3 database.

max_database_size() → int

Return maximum size.

max_length() → int

Return the max size supported by the current implementation of SQLite3.

max_page_count() → int

Return the max page count.

page_size() → int

Return the page size.

pragmas() → list[str]

Return the list of pragmas.

set_pragma(*key: str*, *value: str*) → None

set the pragma value.

3.3.5 Commit System

The commit system provides transactional persistence with history tracking.

When to use: Use `CommitDatabase` for versioned persistence with history, branching, and sync. Every change creates a commit, enabling undo/redo and concurrent editing.

Quick Start

```
from dsviper import CommitDatabase, CommitMutableState

# Open versioned database (created with dsm_util.py)
db = CommitDatabase.open("model.cdb")
db.definitions().inject()

# Create mutable state from latest commit
state = db.state(db.last_commit_id())
mutable = CommitMutableState(state)

# Apply mutations via AttachmentMutating interface
mutating = mutable.attachment_mutating()
mutating.set(MYAPP_A_USER, key, document)

# Commit changes → returns new commit ID
commit_id = db.commit_mutations("Add user", mutable)
```

See also

For detailed examples and the Dual-Layer Contract, see *Commit System*.

Path-Based Mutations

Use `Path` with `update()` to modify specific fields without replacing the entire document. This enables multiplayer editing where concurrent changes to different fields merge automatically.

```
from dsviper import CommitDatabase, CommitMutableState, Path

db = CommitDatabase.open("model.cdb")
db.definitions().inject()

# Build a path to a nested field: document.address.city
path_city = Path.from_field("address").field("city").const()

# Create mutable state
state = db.state(db.last_commit_id())
mutable = CommitMutableState(state)
mutating = mutable.attachment_mutating()

# Update only the city field (not the whole document)
mutating.update(MYAPP_A_USER, user_key, path_city, "Paris")

# Commit
db.commit_mutations("Update city", mutable)
```

With `set()`, concurrent edits to the same document cause one to be overwritten. With `update()`, edits to different fields merge automatically—essential for multiplayer editing.

Choosing the Right Class

Use Case	Class	Example
Open versioned database	`CommitDatabase`	`db = CommitDatabase.open(path)`
Read state at specific commit	`CommitState`	`state = db.state(commit_id)`
Prepare mutations	`CommitMutableState`	`mutable = CommitMutableState(state)`
Read documents (get, keys, has)	`AttachmentGetting`	`getting = state.attachment_getting()`
Write documents (set, update)	`AttachmentMutating`	`mutating = mutable.attachment_mutating()`
Inspect commit metadata	`CommitHeader`	`header = db.commit_header(id)`

Core Classes

dsviper.Commit	A class used to represent a commit.
dsviper.CommitDatabase	A Commit database keeps the history of mutations in a DAG of commit.

continues on next page

Table 18 – continued from previous page

dsviper.CommitDatabaseSQLite	A low-level class used to represent the database based on SQLite3 through the CommitDatabasing interface.
dsviper.CommitDatabaseRemote	A low-level class used to represent a remote Commit database through the CommitDatabasing interface.
dsviper.CommitDatabaseServer	A CommitDatabaseServer provide network access to a CommitDatabase.
dsviper.CommitDatabasing	An interface used to abstract the implementation of the persistence layer for a Commit database.

Commit

class dsviper.**Commit**

Bases: object

A class used to represent a commit.

Note: Not directly instantiable.

header() → *CommitHeader*

Return the header.

program() → *ValueProgram* | None

Return the program or None

CommitDatabase

class dsviper.**CommitDatabase**(*commit_databasing*)

Bases: object

A Commit database keeps the history of mutations in a DAG of commit.

Use the static factory method create_in_memory(), create(…), open(…), connect(…) or connect_local(…).

blob(*blob_id:* ValueBlobId) → *ValueBlob*

Return a blob or None.

blob_getting() → *BlobGetting*

Return the BlobGetting interface.

blob_ids() → set[*ValueBlobId*]

Return the set of blob_id of all available blobs.

blob_info(*blob_id:* ValueBlobId) → *BlobInfo*

Return the information of the blob.

blob_infos(*blob_ids: set[*ValueBlobId*]*) → list[*BlobInfo*]

Return a list of BlobInfo.

blob_statistics() → *BlobStatistics*

Return the statistics for blobs.

blob_stream_append(*blob_stream:* BlobStream, *blob:* ValueBlob) → None

Append the blob to the stream.

blob_stream_close(*blob_stream:* BlobStream) → *ValueBlobId*

Return the computed blob_id and close the stream.

blob_stream_create(*blob_layout:* BlobLayout, *size: int*) → *BlobStream*

Return a stream to fill a blob.

children_commit_ids(*commit_id:* ValueCommitId) → set[*ValueCommitId*]

Return a set of commit_id for the children of the commit.

close() → None

Close the database.

codec_name() → str

Return the name of the codec.

commit(*commit_id:* ValueCommitId) → *Commit*

Return the commit.

commit_databasing() → *CommitDatabasing*

Return the CommitDatabasing interface.

commit_exists(*commit_id:* ValueCommitId) → bool

Return True if the commit exists.

commit_header(*commit_id:* ValueCommitId) → *CommitHeader*

Return the header associated with the commit.

commit_ids() → set[*ValueCommitId*]

Return a set of commit_id for all available commits.

commit_mutations(*label: str*, *commit_mutable_state:* CommitMutableState) → *ValueCommitId*

Create a new commit and return the commit_id.

static connect(*host: str*, *service: str = '54321'*) → *CommitDatabase*

Connect to a remote commit server.

static connect_local(*socket_path: str*) → *CommitDatabase*

Connect to a socket located at socket_path.

static create(*file_path: str*, *, *documentation: str | None = None*) → *CommitDatabase*

Create a database.

create_blob(*blob_layout:* BlobLayout, *blob:* ValueBlob) → *ValueBlobId*

Compute and return the blob_id for a blob and the layout.

create_blob_from_buffer(*buffer*) → *ValueBlobId*

create_blob_buffer($self, buffer) –

Compute and return the blob_id for a object implementing the buffer protocol.

static create_in_memory() → *CommitDatabase*

Create a database in memory.

definitions() → *DefinitionsConst*

Return the definitions.

definitions_hexdigest() → str

Return the hexdigest of the definitions.

delete_commit(*commit_id:* ValueCommitId) → None

Delete a commit.

WARNING: It's not a feature!, it's a trick used during interactive presentation.

disable_commit(*label: str*, *parent_commit_id:* ValueCommitId, *disabled_commit_id:* ValueCommitId) → *ValueCommitId*

Return the commit_id of a new commit that disables another commit.

documentation() → str

Return the documentation.

enable_commit(*label: str*, *parent_commit_id:* ValueCommitId, *enabled_commit_id:* ValueCommitId) → *ValueCommitId*

Return the commit_id of the new commit that enables another commit.

enabled_by_commit_id(*commit_id:* ValueCommitId) → dict[*ValueCommitId*, bool]

Return a dict[ValueCommitId, bool].

extend_definitions(*other:* DefinitionsConst) → *DefinitionsExtendInfo*

Extend the definitions.

fast_forward(*commit_id:* ValueCommitId) → *ValueCommitId*

Return the commit_id of the most plausible head.

first_commit_id() → *ValueCommitId* | None

Return the commit_id of the first commit or None.

forward(*commit_id:* ValueCommitId) → *ValueCommitId*

Return the commit_id of the uniq head or fast_forward.

head_commit_ids() → set[*ValueCommitId*]

Return a set of commit_id for the available heads.

in_memory() → bool

Return True if the database is in memory.

initial_state() → *CommitState*

Return the initial state without any mutations.

is_ancestor(*commit_id:* ValueCommitId, *descendant_id:* ValueCommitId) → bool

Return True if commit_id is a descendant of descendant_id.

is_closed() → bool

Return True is the database is closed.

static is_compatible(*file_path: str*) → bool

Return True if the SQL schema contains the required tables.

is_mergeable(*parent_commit_id:* ValueCommitId, *merged_commit_id:* ValueCommitId) → bool

Return True if a merge is valid.

last_commit_id() → *ValueCommitId* | None

Return the commit_id of the last commit or None.

merge_commit(*label: str*, *parent_commit_id:* ValueCommitId, *merged_commit_id:* ValueCommitId) → *ValueCommitId*

Return the commit_id of the new commit that merges another commit.

nephew_commit_ids(*commit_id:* ValueCommitId) → set[*ValueCommitId*]

Return a set of commit_id for the nephew of the commit.

static open(*file_path: str*, *readonly: bool = False*) → *CommitDatabase*

Open a database.

path() → str

Return the path.

read_blob(*blob_id:* ValueBlobId, *size: int*, *offset: int*) → *ValueBlob*

Return a region of the blob.

reduce_heads() → *ValueCommitId* | None

Reduce to a single head by iteratively merging heads, starting from the last_commit_id and return the new commit_id or None.

reset_commits() → None

Remove all commits except the first one.

WARNING: It's not a feature!, it's a trick used during interactive presentation.

state(*commit_id:* ValueCommitId) → *CommitState*

Return a new state for a commit.

stream_codec_instancing() → *StreamCodecInstancing*

Return the StreamCodecInstancing interface used to encode the binary data.

uuid() → *ValueUUId*

Return the uuid.

CommitDatabaseSQLite

class dsviper.**CommitDatabaseSQLite**

Bases: `object`

A low-level class used to represent the database based on SQLite3 through the CommitDatabasing interface. Use the high-level static factory method CommitDatabase.create(…) or CommitDatabase.open(…).

Note: Not directly instantiable.

begin_transaction() → None

Create a transaction to boost the creation of many blobs.

close() → None

Close the database.

commit() → None

Commit the transaction.

commit_databasing() → *CommitDatabasing*

Return the CommitDatabasing interface.

static create(*file_path: str*, *, *documentation: str* | *None = None*) → *CommitDatabaseSQLite*

Create a database.

static create_in_memory() → *CommitDatabaseSQLite*

Create a database in memory.

in_transaction() → bool

Return True if a transaction is running.

is_closed() → bool

Return True if the database is closed.

static is_compatible(*file_path: str*) → bool

Return True if the SQL schema contains the required tables.

static open(*file_path: str*, *readonly: bool = False*) → *CommitDatabaseSQLite*

Open a database.

rollback() → None

Roll back the transaction.

sqlite() → *SQLite*

Return a SQLite.

CommitDatabaseRemote

class dsviper.**CommitDatabaseRemote**

Bases: object

A low-level class used to represent a remote Commit database through the CommitDatabasing interface. Use the high-level static factory method CommitDatabase.connect(…) or CommitDatabase.connect_local(…).

Note: Not directly instantiable.

close() → None

Close the connection.

commit_databasing() → *CommitDatabasing*

Return the CommitDatabasing interface.

static connect(*host: str*, *service: str = '54321'*) → *CommitDatabaseRemote*

Connect to a remote commit server.

static connect_local(*socket_path: str*) → *CommitDatabaseRemote*

Connect to the socket located at socket_path.

download_speed(*size: int = 1*) → float

Return the download speed in MBps (Mega Bytes per second) The size is expressed in Mega Bytes [1-1000].

is_closed() → bool

Return True if the server is closed.

ping() → float

Ping (latency is the technically more correct term) means the time it takes for a small data set to be transmitted from your device to a server and back to your device again. The ping time is measured in milliseconds (ms).

upload_speed(*size: int = 1*) → float

Return the upload speed in MBps (Mega Bytes per second) The size is expressed in Mega Bytes [1-1000].

CommitDatabaseServer

class dsviper.**CommitDatabaseServer**(*database_path*, *socket*, *logging*, *cancelation*)

Bases: object

A CommitDatabaseServer provide network access to a CommitDatabase.

finish() → None

block until all client thread handle the cancelation request.

start() → None

Open a database.

step(*timeout_in_sec: int = 1*) → bool

accept new connection in a non blocking way.

CommitDatabasing

class dsviper.**CommitDatabasing**

Bases: object

An interface used to abstract the implementation of the persistence layer for a Commit database. This is a low-level interface (a driver) and requires a deep understanding of Commit to be used correctly.

Note: Not directly instantiable.

begin_transaction(*mode: str | None = None*) → None

Begin a transaction where the mode is 'Deferred', 'Immediate' or 'Exclusive'.

blob(*blob_id:* ValueBlobId) → *ValueBlob* | None

Return a blob or None.

blob_datas(*blob_ids: set[*ValueBlobId*]*) → list[*BlobData*]

Return a list of BlobData.

blob_ids() → set[*ValueBlobId*]

Return a set of blob_id for all available blobs.

blob_info(*blob_id:* ValueBlobId) → *BlobInfo* | None

Return the information of a blob.

blob_infos(*blob_ids: set[*ValueBlobId*]*) → list[*BlobInfo*]

Return a list of BlobInfo.

blob_statistics() → *BlobStatistics*

Return the statistics for blobs.

blob_stream_close(*stream_id:* ValueUUId, *blob_id:* ValueBlobId) → None

Close the stream.

blob_stream_create(*blob_layout:* BlobLayout, *size: int*) → *ValueUUId*

Return the uuid of the stream.

blob_stream_delete(*stream_id:* ValueUUId) → None

Delete the stream.

blob_stream_write(*stream_id:* ValueUUId, *blob:* ValueBlob, *offset: int*) → None

Write a region of the blob.

children_commit_ids(*commit_id:* ValueCommitId) → set[*ValueCommitId*]

Return the set of commit_id for the children of the commit.

close() → None

Close the connection.

codec_name() → str

Return the name of the codec.

commit() → None

Commit the transaction.

commit_data(*commit_id:* ValueCommitId) → *CommitData* | None

Return a CommitData or None.

commit_datas(*commit_ids: set[*ValueCommitId*]* | *None* = *None*) → list[*CommitData*]

Return the list of CommitData.

commit_exists(*commit_id:* ValueCommitId) → bool

Return True if the commit exists.

commit_header(*commit_id:* ValueCommitId) → *CommitHeader*

Return the header of the commit associated with the commit_id.

commit_ids() → set[*ValueCommitId*]

Return the set of commit_ids.

create_blob(*blob_id:* ValueBlobId, *blob_layout:* BlobLayout, *blob:* ValueBlob) → bool

Create a blob.

create_blobs(*blob_datas: list[*BlobData*]*) → set[*ValueBlobId*]

Create blobs from a list of BlobDatas and return a set of blob_id.

create_commit_data(*commit_data:* CommitData) → bool

Create a commit if not present or return False.

create_zero_blob(*blob_id:* ValueBlobId, *blob_layout:* BlobLayout, *size: int*) → bool

Return the True if the blob was created.

data_version() → int

Return the data version.

definitions() → *DefinitionsConst*

Return the definitions.

definitions_hexdigest() → str

Return the hexdigest of the definitions.

delete_commit(*commit_id:* ValueCommitId) → None

Delete a commit.

WARNING: It's not a feature!, it's a trick used during interactive presentation.

documentation() → str

Return the documentation.

extend_definitions(*other:* DefinitionsConst) → *DefinitionsExtendInfo*

Extend the definitions.

first_commit_id() → *ValueCommitId* | None

Return the commit_id of the first commit or None.

freeze_blob(*blob_id:* ValueBlobId) → bool

Return True if the blob was frozen.

head_commit_ids() → set[*ValueCommitId*]

Return the set of commit_id for the heads.

in_transaction() → bool

Return True if a transaction is running.

is_closed() → bool

Return True is the connection to the database is closed.

last_commit_id() → *ValueCommitId* | None

Return the commit_id of the last commit or None.

nephew_commit_ids(*commit_id:* ValueCommitId) → set[*ValueCommitId*]

Return the set of commit_id for the nephew of the commit.

path() → str

Return the path.

read_blob(*blob_id:* ValueBlobId, *size: int*, *offset: int*) → *ValueBlob*

Return the blob at offset.

reset_commits() → None

Remove all commits except the first one.

WARNING: It's not a feature!, it's a trick used during interactive presentation.

rollback() → None

Roll back the transaction.

sync_data(*commit_ids: set[*ValueCommitId*]*) → *CommitSyncData*

Return a CommitSyncData.

unknown_blob_ids(*blob_ids: set[*ValueBlobId*]*) → set[*ValueBlobId*]

Return a set of blob_id for all unknown blobId found in blob_ids.

uuid() → *ValueUUId*

Return the uuid.

write_blob(*blob_id:* ValueBlobId, *blob:* ValueBlob, *offset: int*) → None

Write a blob at offset.

State Access

dsviper.CommitState	A class used to represent data at a specific commit.
dsviper.CommitMutableState	A class used to register the mutations of a value executed through the attachmentMutating interface.

CommitState

class dsviper.**CommitState**

Bases: object

A class used to represent data at a specific commit. Use the attachmentGetting interface to retrieve value by key from attachments.

Note: Not directly instantiable.

attachment_getting() → *AttachmentGetting*
Return the AttachmentGetting interface.

cache_hit_rate() → float
Return the cache hit rate.

cache_hits() → int
Return the count of value return from the cache.

cache_preload() → float
Return the tuple time to get all documents.

cache_requests() → int
Return the count of value requested.

commit_id() → *ValueCommitId*
Return the identifier of the commit.

commit_state_tracing() → *CommitStateTracing*
Return the AttachmentGetting interface.

definitions() → *DefinitionsConst*
Return the definitions.

eval_actions() → list[*CommitEvalAction*]
Return a list of CommitEvalAction.

traced_opcodes() → list[*ValueOpcode*]
Return the list of opcodes from the commit trace.

CommitMutableState

class dsviper.**CommitMutableState**(*commit_state*)
Bases: object

A class used to register the mutations of a value executed through the attachmentMutating interface.

attachment_getting() → *AttachmentGetting*
Return the AttachmentGetting interface.

attachment_mutating() → *AttachmentMutating*
Return the attachmentMutating interface.

commit_state() → *CommitState*
Return the CommitState.

commit_state_tracing() → *CommitStateTracing*
Return the AttachmentGetting interface.

mutations() → *ValueProgram*
Return the program for the current mutations.

History & DAG

dsviper.CommitNode	A class used to represent a node in the commit DAG.

continues on next page

Table 20 – continued from previous page

dsviper.CommitNodeGrid	A class used to represent the location of a commit node in the grid.
dsviper.CommitNodeGridBuilder	A class used to build the grid layout of the commit DAG.
dsviper.CommitHeader	A class used to represent the header of a commit.
dsviper.CommitData	A class used to represent data associated with a commit during the synchronization of two databases.
dsviper.CommitEvalAction	A class used by the evaluator to reconstruct a value by executing mutation opcodes.

CommitNode

class dsviper.**CommitNode**

Bases: object

A class used to represent a node in the commit DAG.

Note: Not directly instantiable.

static build(*commit_database:* CommitDatabase) → *CommitNode*

Create and return the DAG of commit.

children() → list[*CommitNode*]

Return the list of childs.

header() → *CommitHeader*

Return the header.

CommitNodeGrid

class dsviper.**CommitNodeGrid**

Bases: object

A class used to represent the location of a commit node in the grid.

Note: Not directly instantiable.

child_count() → int

Return the number of children.

children() → list[*CommitNodeGrid*]

Return the list of childs.

column() → int

Return the column.

commit_id() → *ValueCommitId*

Return the commitId.

has_children() → bool

Return true if the node have children.

header() → *CommitHeader*

Return the header.

parent() → *CommitNodeGrid* | None

Return the parent or None.

row() → int

Return the row.

CommitNodeGridBuilder

class dsviper.**CommitNodeGridBuilder**

Bases: object

A class used to build the grid layout of the commit DAG.

Note: Not directly instantiable.

static build(*commit_node:* CommitNode) → *CommitNodeGridBuilder*

Create and return the builder.

column_max() → int

Return the max index for a column.

nodes() → dict[*ValueCommitId*, *CommitNodeGrid*]

Return a dict[uuid, CommitNodeGrid].

root() → *CommitNodeGrid* | None

Return the root node or None.

row_max() → int

Return the max index for row.

CommitHeader

class dsviper.**CommitHeader**

Bases: object

A class used to represent the header of a commit.

Note: Not directly instantiable.

commit_id() → *ValueCommitId*

Return the commit_id of the commit.

commit_type() → str

Return the type ('Mutations', 'Disable', 'Enable' or 'Merge').

label() → str

Return the label.

parent_commit_id() → *ValueCommitId*

Return the commit_id of the parent commit.

target_commit_id() → *ValueCommitId*

Return the commit_id of the target commit.

timestamp() → float

Return the timestamp.

CommitData

class dsviper.**CommitData**

Bases: object

A class used to represent data associated with a commit during the synchronization of two databases.

Note: Not directly instantiable.

blob_ids(*stream_codec_instancing:* StreamCodecInstancing, *definitions:* DefinitionsConst) → set[*ValueBlobId*]

Return the set of blob_id referenced by the opcodes.

data() → *CommitData*

Return the blob of the encoded opcodes.

header() → *CommitHeader*

Return the header.

opcodes(*stream_codec_instancing:* StreamCodecInstancing, *definitions:* DefinitionsConst) → list[*ValueOpcode*]

Return the list of opcodes.

static sort(*commit_datas: list[*CommitData*]*) → list[*CommitData*]

Return the list of CommitDatas topologically sorted.

CommitEvalAction

class dsviper.**CommitEvalAction**

Bases: object

A class used by the evaluator to reconstruct a value by executing mutation opcodes.

Note: Not directly instantiable.

enabled() → bool

Return True if the commit is enabled.

header() → *CommitHeader*

Return the header of the commit.

program() → *ValueProgram*

Return the program.

Synchronization

dsviper.CommitStore	A high-level application class used to implement the store, dispatch, undo/redo and notification concepts inspired by the redux approach.
dsviper.CommitStoreNotifying	An interface used to represent the notification emitted by a store.
dsviper.CommitSynchronizer	A class used to synchronize two concrete databases through the CommitDatabasing interface (low-level driver interface).
dsviper.CommitSynchronizerInfo	A class used to represent data exchanged during the synchronization of two databases.

continues on next page

Table 21 – continued from previous page

dsviper.CommitSynchronizerInfoTransmit	A class used to represent statistics of data exchanged during the synchronization of two databases.
dsviper.CommitSyncData	A class used to represent data exchanged during the synchronization of two databases.

CommitStore

class dsviper.**CommitStore**

Bases: `object`

A high-level application class used to implement the store, dispatch, undo/redo and notification concepts inspired by the redux approach.

The implementation used a Commit database for the persistence layer.

attachment_getting() → *AttachmentGetting*

Return the AttachmentGetting interface of the current state or raise.

can_redo() → bool

Return True if the store can redo.

can_undo() → bool

Return True if the store can undo.

clear_undo_redo() → None

clear the undo redo stack.

close() → None

Close the database.

commit_mutations(*label: str*, *commit_mutable_state:* CommitMutableState) → None

Commit the mutations tracked by the mutable state.

database() → *CommitDatabase*

Return the database or raise.

definitions() → *DefinitionsConst*

Return the definitions.

delete_commit(*commit_id:* ValueCommitId) → None

Delete a commit.

WARNING: It's not a feature!, it's a trick used during interactive presentation.

disable_commit(*commit_id:* ValueCommitId) → None

Disable a commit.

dispatch(*label: str*, *callable: Callable*, **args*) → None

Dispatch a callable.

dispatch_diff(*label: str*, *attachment:* Attachment, *key:* ValueKey, *value: _InputValues*, *, *recursive: bool* = *False*) → None

Dispatch a mutating diff(…).

dispatch_enable_commit(*commit_id:* ValueCommitId, *enabled: bool*) → None

Dispatch the act of enabling/disabling a commit.

dispatch_set(*label: str*, *attachment:* Attachment, *key:* ValueKey, *value: _InputValues*) → None

Dispatch a mutating set(…).

dispatch_update(*label: str*, *attachment:* Attachment, *key:* ValueKey, *path:* PathConst, *value: _InputValues*) → None

Dispatch a mutating update(…).

enable_commit(*commit_id:* ValueCommitId) → None

Enable a commit.

extend_definitions(*definitions:* DefinitionsConst) → *DefinitionsExtendInfo*

Extend the database definitions and notify.

forward() → None

Move to the most plausible head.

has_database() → bool

Return true there is a database.

has_state() → bool

Return true there is a state.

static instance() → *CommitStore*

Return the singleton.

merge_commit(*commit_id:* ValueCommitId) → None

Merge a commit.

mutable_state() → *CommitMutableState*

Return a new mutable state for the current state.

notifier() → *CommitStoreNotifying* | None

Return the notifier or None.

notify_database_did_close() → None

Send a notification to inform that the database was closed.

notify_database_did_open() → None

Send a notification to inform that the database was opened.

notify_database_did_reset() → None

Send a notification to inform that the database did reset.

notify_database_will_reset() → None

Send a notification to inform that the database will reset.

notify_definitions_did_change() → None

Send a notification to inform that the definitions has changed.

notify_dispatch_error(*error:* Error) → None

Send a notification to inform that an error occurred during a dispatch.

notify_reset_database() → None

Send a notification to reset the database.

notify_state_did_change() → None

Send a notification to inform that the current state has changed.

notify_stop_live() → None

Send a notification to stop the live mode.

redo() → None

Implement the undo idiom for a Commit database.

reduce_heads() → None

Reduce to a single head by iteratively merging heads, starting from the last_commit_id.

reset() → None

Remove all commits except the first one.

WARNING: It's not a feature!, it's a trick used during interactive presentation.

reset_undo_redo() → None

Reset the undo redo stack.

set_database(*commit_database:* CommitDatabase) → None

set the database.

set_notifier(*notifier:* CommitStoreNotifying) → None

Set the notifier.

set_state(*commit_state:* CommitState) → None

set the state.

state() → *CommitState* | None

Return the current state or raise.

undo() → None

Implement the undo idiom for a Commit database.

undo_stack_ids() → tuple[list[*ValueCommitId*], int | None]

Return (commit_ids, current_commit_id).

use(*commit_database:* CommitDatabase) → None

Use a Commit database.

use_commit(*commit_id:* ValueCommitId) → None

Use a commit.

CommitStoreNotifying

class dsviper.**CommitStoreNotifying**

Bases: object

An interface used to represent the notification emitted by a store.

Note: Not directly instantiable.

static create(*object*) → *CommitStoreNotifying*

Return a new CommitStoreStoringBaseNotifying if the Python object responds to the interface or raise.

notify_database_did_close() → None

Send a notification to inform that the database was closed.

notify_database_did_open() → None

Send a notification to inform that the database was opened.

notify_database_did_reset() → None

Send a notification to inform that the database did reset.

notify_database_will_reset() → None

Send a notification to inform that the database will be reset.

notify_definitions_did_change() → None

Send a notification to inform that the definitions have changed.

notify_dispatch_error(*error:* Error) → None

Send a notification to inform that an error occurred during a dispatch.

notify_reset_database() → None

Send a notification to reset the database.

notify_state_did_change() → None

Send a notification to inform that the current state has changed.

notify_stop_live() → None

Send a notification to stop the live mode.

CommitSynchronizer

class dsviper.**CommitSynchronizer**(*source*, *target*, *mode='Sync'*, *size_of_packed_blobs=25000000*)

Bases: object

A class used to synchronize two concrete databases through the CommitDatabasing interface (low-level driver interface).

MODE_FETCH = 'Fetch'

MODE_PUSH = 'Push'

MODE_SYNC = 'Sync'

mode() → str

Return the mode.

size_of_packed_blobs() → int

Return the size of the blob used to pack smaller blobs together.

source() → *CommitDatabasing*

Return the CommitDatabasing interface for the source.

sync(*logging:* Logging) → *CommitSynchronizerInfo*

Synchronize the content of the source and the target.

target() → *CommitDatabasing*

Return the CommitDatabasing interface for the target.

CommitSynchronizerInfo

class dsviper.**CommitSynchronizerInfo**

Bases: object

A class used to represent data exchanged during the synchronization of two databases.

Note: Not directly instantiable.

fetch() → *CommitSynchronizerInfoTransmit*

Return information for fetched resources.

need_transmit() → bool

Return True if the synchronization requires exchanging data.

push() → *CommitSynchronizerInfoTransmit*

Return information for pushed resources.

updated_definitions() → bool

Return True if the definitions has updated.

CommitSynchronizerInfoTransmit

class dsviper.**CommitSynchronizerInfoTransmit**

Bases: object

A class used to represent statistics of data exchanged during the synchronization of two databases.

Note: Not directly instantiable.

blob_bytes() → int

Return the number of bytes for blobs.

blobs() → int

Return the number of blobs.

commit_bytes() → int

Return the number of commit bytes fetched.

commits() → int

Return the number of commits fetched.

extend_info() → *DefinitionsExtendInfo*

Return the information to extend.

CommitSyncData

class dsviper.**CommitSyncData**

Bases: object

A class used to represent data exchanged during the synchronization of two databases.

Note: Not directly instantiable.

codec_name() → str

Return the name of the codec.

commit_datas() → list[*CommitData*]

Return the list of CommitData.

definitions_hexdigest() → str

Return the hexdigest of the definitions.

Tracing

dsviper.CommitStateTracing	An interface used to trace a value (aka document).
dsviper.CommitStateTrace	A class used to represent traced opcode.
dsviper.CommitStateTraceProgram	A class used to represent traced opcode.
dsviper.ValueProcessorTrace	A class used to represent traced opcode.
dsviper.ValueProcessorTraceOpcode	A class used to represent a traced opcode.

CommitStateTracing

class dsviper.**CommitStateTracing**

Bases: object

An interface used to trace a value (aka document).

Note: Not directly instantiable.

trace(*attachment:* Attachment, *key:* ValueKey) → *CommitStateTrace*

Return an CommitStateTrace associated with the key in the attachment.

CommitStateTrace

class dsviper.**CommitStateTrace**

Bases: object

A class used to represent traced opcode.

Note: Not directly instantiable.

attachment() → *Attachment*

Return the attachment.

key() → *ValueKey*

Return the key.

programs() → list[*CommitStateTraceProgram*]

Return the list of programs.

value() → *ValueOptional*

Return the value.

CommitStateTraceProgram

class dsviper.**CommitStateTraceProgram**

Bases: object

A class used to represent traced opcode.

Note: Not directly instantiable.

header() → *CommitHeader* | None

Return the commit header or None.

trace() → *ValueProcessorTrace*

Return the trace.

ValueProcessorTrace

class dsviper.**ValueProcessorTrace**

Bases: object

A class used to represent traced opcode.

Note: Not directly instantiable.

enabled() → bool

Return the enabled.

opcodes() → list[*ValueProcessorTraceOpcode*]

Return the list of traces opcodes.

ValueProcessorTraceOpcode

class dsviper.**ValueProcessorTraceOpcode**

Bases: object

A class used to represent a traced opcode.

Note: Not directly instantiable.

exception() → str | None

Return the exception or None.

opcode() → *ValueOpcode*

Return the opcode.

3.3.6 Binary Data (Blobs)

Blobs provide efficient binary data storage with typed layouts and zero-copy NumPy integration.

When to use: Use blobs for binary data like images, meshes, and raw buffers. `BlobArray` for typed arrays, `BlobPack` for structured multi-region data.

Quick Start

```
from dsviper import BlobLayout, BlobArray, BlobPack, BlobPackDescriptor
import numpy as np

# Typed array: 100 vec3 positions (float, 3 components)
layout = BlobLayout('float', 3)
positions = BlobArray(layout, 100)

# Zero-copy NumPy access
np_view = np.array(positions, copy=False)
np_view[0] = [1.0, 2.0, 3.0]  # Modifies original

# Structured blob: mesh with multiple regions
desc = BlobPackDescriptor()
desc.add_region('positions', BlobLayout('float', 3), 100)
desc.add_region('normals', BlobLayout('float', 3), 100)
desc.add_region('indices', BlobLayout('uint', 3), 50)
mesh = BlobPack(desc)
```

(continues on next page)

(continued from previous page)

```
# Access region as NumPy array
pos_np = np.array(mesh['positions'], copy=False)
```

Choosing the Right Class

Data Type	Class	Example
Raw bytes	`ValueBlob`	`ValueBlob(bytes_data)`
Typed array	`BlobArray`	`BlobArray(layout, count)`
Multiple regions	`BlobPack`	Mesh with pos/normals/indices
Database reference	`ValueBlobId`	SHA-1 hash reference
Large files (>2GB)	`BlobStream`	Streaming upload

Core Classes

dsviper.BlobLayout	A class used to describe the layout of the element in a blob.
dsviper.BlobArray	A class used to reinterpret the blob of the internal BlobView as an array<blob_layout.data_type>.
dsviper.BlobPack	A class used to implements memory regions in one blob.
dsviper.BlobPackDescriptor	A class used to describe binary regions.
dsviper.BlobPackRegion	A class used for a region that implement the buffer protocol.

BlobLayout

class dsviper.**BlobLayout**(*data_type='uchar'*, *components=1*)

Bases: `object`

A class used to describe the layout of the element in a blob. data_type is 'uchar', 'ushort', 'uint', 'ulong', 'char', 'short', 'int', 'long', 'half', 'float' or 'double', and components is limited to 255. The default layout is 'uchar-1' (aka a BLOB).

byte_count() → int

Return the size of an element in bytes.

components() → int

Return the number of packed values in an element.

data_type() → int

Return the data type.

data_type_byte_count() → int

Return the size of the data type in bytes.

data_type_representation() → str

Return the data_type representation.

static parse(*representation: str*) → *BlobLayout*

Return a BlobLayout by parsing the string '<data_type>-<components>' ex: 'uchar-1', 'float-2', 'float-3', 'half-4', 'double-16'.

representation() → str

Return a string representation.

BlobArray

class dsviper.**BlobArray**(*blob_layout*, *size*)

Bases: object

A class used to reinterpret the blob of the internal BlobView as an array<blob_layout.data_type>.

blob() → *ValueBlob*

Return the blob.

blob_layout() → *BlobLayout*

Return the blob layout.

blob_view() → *BlobView*

Return the BlobView.

static from_blob(*blob_layout:* BlobLayout, *blob:* ValueBlob) → *BlobArray*

Return a BlobBuffer from a layout and a blob.

BlobPack

class dsviper.**BlobPack**(*descriptor*)

Bases: object

A class used to implements memory regions in one blob.

blob() → *ValueBlob*

Return the blob used to encode the blob pack.

check(*name: str*) → *BlobPackRegion*

Return the BlobPackItem or raise.

static from_blob(*blob:* ValueBlob) → *BlobPack*

Return a BlobPack from a blob or raise.

query(*name: str*) → *BlobPackRegion* | None

Return the BlobPackItem or None.

regions() → list[*BlobPackRegion*]

Return a list of BlobPackItem.

BlobPackDescriptor

class dsviper.**BlobPackDescriptor**

Bases: object

A class used to describe binary regions.

add_region(*name: str*, *blob_layout:* BlobLayout, *count: int*) → None

Add and named region, where count is the number of elements defined by blob_layout type.

BlobPackRegion

class dsviper.**BlobPackRegion**

Bases: object

A class used for a region that implement the buffer protocol.

Note: Not directly instantiable.

blob() → *ValueBlob*

Return the blob used by the BlobPack.

blob_layout() → *BlobLayout*

Return the blob layout.

byte_count() → int

Return the number of byte.

copy(*buffer: Buffer*) → None

Copy the content of the buffer to the region.

count() → int

Return the number of element.

data_count() → int

Return the number of data.

name() → str

Return the name.

offset() → int

Return the offset in the blob.

Blob I/O

dsviper.BlobStream	A class used to copy a huge blob (> 2GB) to a database.
dsviper.BlobEncoder	A class used to encode a blob.
dsviper.BlobEncoderLayout	A class used to represent the layout of one element.
dsviper.BlobView	A class used to interpret the bytes of a blob from the layout.
dsviper.BlobData	A class used to represent the data associated with a blob during the synchronization of two databases.

BlobStream

class dsviper.**BlobStream**

Bases: object

A class used to copy a huge blob (> 2GB) to a database. Use the method database.blob_stream_create(blob_layout, size) to create a BlobStream, then the method database.blob_stream_append(…) to incrementally fill the content and finally the method database.blob_stream_close(…) to get the blob_id computed from the immutable content of the blob and its layout.

Note: Not directly instantiable.

blob_id() → *ValueBlobId*

Return the identifier of the blob.

is_closed() → bool

Return True if the stream is closed.

offset() → int

Return the current offset.

remaining() → int

Return the remaining bytes count to fill the blob.

BlobEncoder

class dsviper.**BlobEncoder**(*blob_layout*)

Bases: object

A class used to encode a blob.

blob_layout() → *BlobLayout*

Return the layout of the blob.

encoder_layout() → *BlobEncoderLayout*

Return the encoder layout.

end_encoding() → *ValueBlob*

Return the encoded blob.

write(*value: _InputValues*) → None

Write a value at the end of the blob. The parameter value is an int, a real or a tuple compatible with the layout.

BlobEncoderLayout

class dsviper.**BlobEncoderLayout**

Bases: object

A class used to represent the layout of one element. The blob is interpreted as an immutable vector<blob_layout>.

Note: Not directly instantiable.

blob_layout() → *BlobLayout*

Return the layout.

element_type() → _NumericTypes

Return the type of the element.

type() → _NumericTypes | *TypeVec*

Return the type of the value for an element.

BlobView

class dsviper.**BlobView**(*blob_layout*, *blob*)

Bases: object

A class used to interpret the bytes of a blob from the layout.

blob() → *ValueBlob*

Return the blob.

blob_layout() → *BlobLayout*

Return the blob_layout.

count() → int

Return the number of elements in the blob.

data_count() → int

Return the number of data in the blob.

encoder_layout() → *BlobEncoderLayout*

Return the encoder layout.

BlobData

class dsviper.**BlobData**

Bases: object

A class used to represent the data associated with a blob during the synchronization of two databases.

Note: Not directly instantiable.

blob() → *ValueBlob*

Return the blob.

blob_id() → *ValueBlobId*

Return the identifier the blob.

blob_layout() → *BlobLayout*

Return the layout of the blob.

size() → int

Return the size of the blob.

Database Integration

dsviper.BlobGetting	An interface used to retreive blobs from a persistence layer.
dsviper.BlobInfo	A class used to represent various information of a blob in the persistence layer.
dsviper.BlobStatistics	A class used to represent the statistics about blobs.

BlobGetting

class dsviper.**BlobGetting**

Bases: object

An interface used to retreive blobs from a persistence layer.

Note: Not directly instantiable.

blob(*blob_id:* ValueBlobId) → *ValueBlob*

Return a blob or None.

blob_ids() → set[*ValueBlobId*]

Return the set of blob_id.

blob_info(*blob_id:* ValueBlobId) → *BlobInfo*

Return the info for the blob.

blob_infos(*blob_ids: set[*ValueBlobId*]*) → list[*BlobInfo*]

Return a list of BlobInfo.

blob_statistics() → *BlobStatistics*

Return the statistics for blobs.

BlobInfo

class dsviper.**BlobInfo**

Bases: object

A class used to represent various information of a blob in the persistence layer.

Note: Not directly instantiable.

blob_id() → *ValueBlobId*

Return the identifier.

blob_layout() → *BlobLayout*

Return the layout.

chunked() → bool

Return True if the blob is greater than 2GB and requires the BlobIO API.

row_id() → int

Return the SQLite3 rowid.

size() → int

Return the size.

BlobStatistics

class dsviper.**BlobStatistics**

Bases: object

A class used to represent the statistics about blobs.

Note: Not directly instantiable.

count() → int

Return the number of blobs.

max_size() → int

Return the size of the biggest blob in bytes.

min_size() → int

Return the size of the smallest blob in bytes.

total_size() → int

Return the total size in bytes.

3.3.7 Serialization

Serialization classes provide binary and token-based encoding/decoding.

When to use: Use `Value.encode()` / `Value.decode()` for binary serialization. Use streams for low-level I/O and custom protocols.

Quick Start

```
from dsviper import Value, ValueString, Type, Definitions

# Create a value
value = ValueString("hello")

# Encode to binary blob
blob = Value.encode(value)

# Decode from binary (requires type and definitions)
defs = Definitions()
decoded = Value.decode(blob, Type.STRING, defs.const())

# Works with any type
from dsviper import TypeVector, ValueVector
t_vec = TypeVector(Type.INT64)
vec = Value.create(t_vec, [1, 2, 3])
blob = Value.encode(vec)
decoded = Value.decode(blob, t_vec, defs.const())
```

JSON Serialization

For web integration and REST APIs, use `Value.json_encode()` and `Value.json_decode()`.

```
from dsviper import Value, ValueString, Type, Definitions

# Encode primitive to JSON
value = ValueString("hello")
json_str = Value.json_encode(value)  # '"hello"'

# Decode JSON (requires type and definitions)
defs = Definitions()
decoded = Value.json_decode(json_str, Type.STRING, defs.const())
```

Flask REST API Example

Real-world pattern for exposing Viper data via REST:

```
from flask import Flask, make_response
from dsviper import CommitDatabase, Value

app = Flask(__name__)

@app.get("/material/<uuid:instance_id>")
def material_api(instance_id):
    db = CommitDatabase.open("model.cdb")
```

(continues on next page)

(continued from previous page)

```
    doc = db.state(db.last_commit_id()).attachment_getting().get(
        MYAPP_A_Material, ValueKey.create(MYAPP_C_Material, str(instance_id))
    )
    db.close()

    # Encode document to JSON with indentation
    json_string = Value.json_encode(doc, indent=2)
    res = make_response(json_string)
    res.mimetype = "application/json"
    return res
```

DSM schema can also be exported for client-side validation:

```
@app.get("/schema")
def schema_api():
    db = CommitDatabase.open("model.cdb")
    dsm_defs = db.definitions().to_dsm_definitions()
    db.close()

    json_string = dsm_defs.json_encode(indent=2)
    res = make_response(json_string)
    res.mimetype = "application/json"
    return res
```

Choosing the Right Approach

Use Case	API	Note
Binary serialization	`Value.encode()`	Default codec
Binary deserialization	`Value.decode()`	Requires type + definitions
JSON for web APIs	`Value.json_encode()`	REST/web integration
Custom codec	`Codec.STREAM_*`	Raw, binary, token formats
File I/O	`StreamWriterFile`	Low-level streaming

Codec

dsviper.Codec	A class used to instantiate a codec.
dsviper.StreamCodecInstancing	An interface used to create encoder, decoder and sizer.

Codec

class dsviper.**Codec**

Bases: `object`

A class used to instantiate a codec. Use the static factory method check(…) and query(…).

Note: Not directly instantiable.

STREAM_BINARY = StreamBinary

STREAM_RAW = StreamRaw

STREAM_TOKEN_BINARY = StreamTokenBinary

static check(*name: str*) → *StreamCodecInstancing*

Return the StreamCodecInstancing interface of the codec or raise.

static query(*name: str*) → *StreamCodecInstancing*

Return the StreamCodecInstancing interface of the codec or None.

StreamCodecInstancing

class dsviper.**StreamCodecInstancing**

Bases: object

An interface used to create encoder, decoder and sizer.

Note: Not directly instantiable.

create_decoder(*blob:* ValueBlob) → *StreamDecoding*

Create and return a decoder.

create_encoder() → *StreamEncoding*

Create and return an encoder.

create_sizer() → *StreamSizing*

Create and return a sizer.

name() → str

Return the name.

Binary Streams

dsviper.StreamBinaryReader	A class used to read data through the StreamRawReading interface (handle endianness).
dsviper.StreamBinaryWriter	A class used to write data through the StreamRawWriting interface (handle endianness).
dsviper.StreamTokenBinaryReader	A class used to read through the StreamRawReading interface (handle endianness).
dsviper.StreamTokenBinaryWriter	A class used to write through the StreamRawWriting interface (handle endianness).

StreamBinaryReader

class dsviper.**StreamBinaryReader**(*stream_raw_reading*)

Bases: object

A class used to read data through the StreamRawReading interface (handle endianness).

read_blob() → *ValueBlob*

Read and return a blob.

read_blob_id() → *ValueBlobId*

Read and return a blob_id.

read_bool() → bool

Read and return a Python bool.

read_commit_id() → *ValueCommitId*

Read and return a commit_id.

read_double() → float

Read and return a Python float.

read_doubles(*size: int*) → *ValueVec*

Read an array of doubles and return a vec<double, size>.

read_float() → float

Read and return a Python float.

read_floats(*size: int*) → *ValueVec*

Read an array of float and return a vec<float, size>.

read_int16() → int

Read and return a Python int.

read_int16s(*size: int*) → *ValueVec*

Read an array of int16 and return a vec<int16, size>.

read_int32() → int

Read and return a Python int.

read_int32s(*size: int*) → *ValueVec*

Read an array of int32 and return a vec<int32, size>.

read_int64() → int

Read and return a Python int.

read_int64s(*size: int*) → *ValueVec*

Read an array of int64 and return a vec<int64, size>.

read_int8() → int

Read and return a Python int.

read_int8s(*size: int*) → *ValueVec*

Read an array of int8 and return a vec<int8, size>.

read_string() → str

Read and return a Python str.

read_uint16() → int

Read and return a Python int.

read_uint16s(*size: int*) → *ValueVec*

Read an array of uint16 and return a vec<uint16, size>.

read_uint32() → int

Read and return a Python int.

read_uint32s(*size: int*) → *ValueVec*

Read an array of uint32 and return a vec<uint32, size>.

read_uint64() → int

Read and return a Python int.

read_uint64s(*size: int*) → *ValueVec*

Read an array of uint64 and return a vec<uint64, size>.

read_uint8() → int

Read and return a Python int.

read_uint8s(*size: int*) → *ValueVec*

Read an array of uint8 and return a vec<uint8, size>.

read_uuid() → *ValueUUId*

Read and return uuid.

stream_reading() → *StreamReading*

Return the StreamReading interface.

StreamBinaryWriter

class dsviper.**StreamBinaryWriter**(*stream_raw_reading*)

Bases: object

A class used to write data through the StreamRawWriting interface (handle endianness).

stream_writing() → *StreamWriting*

Return the StreamWriting interface.

write_blob(*value:* ValueBlob) → None

Write a blob.

write_blob_id(*value:* ValueBlobId) → None

Write a blob_id.

write_bool(*value: bool*) → None

Write a bool.

write_commit_id(*value:* ValueCommitId) → None

Write a commit_id.

write_double(*value: float*) → None

Write a double.

write_doubles(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of double.

write_float(*value: float*) → None

Write a float.

write_floats(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of float.

write_int16(*value: int*) → None

Write an int16.

write_int16s(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of int16.

write_int32(*value: int*) → None

Write an int32.

write_int32s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int32.

write_int64(*value: int*) → None
Write an int64.

write_int64s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int64.

write_int8(*value: int*) → None
Write an int8.

write_int8s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int8.

write_string(*value: str*) → None
Write a string.

write_uint16(*value: int*) → None
Write an uint16.

write_uint16s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint16.

write_uint32(*value: int*) → None
Write an uint32.

write_uint32s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint32.

write_uint64(*value: int*) → None
Write an uint64.

write_uint64s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint64.

write_uint8(*value: int*) → None
Write an uint8.

write_uint8s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint8.

write_uuid(*value:* ValueUUId) → None
Write an uuid.

StreamTokenBinaryReader

class dsviper.**StreamTokenBinaryReader**(*stream_raw_reading*)
Bases: object

A class used to read through the StreamRawReading interface (handle endianness).

read_blob() → *ValueBlob*
Read and return a blob.

read_blob_id() → *ValueBlobId*
Read and return a blob_id.

read_bool() → bool

Read and return a Python bool.

read_commit_id() → *ValueCommitId*

Read and return a commit_id.

read_double() → float

Read and return a Python float.

read_doubles(*size: int*) → *ValueVec*

Read an array of double and return a vec<double, size>.

read_float() → float

Read aan return a float.

read_floats(*size: int*) → *ValueVec*

Read an array of float and return a vec<float, size>.

read_int16() → int

Read and return a Python int.

read_int16s(*size: int*) → *ValueVec*

Read an array of int16 and return a vec<int16, size>.

read_int32() → int

Read and return a Python int.

read_int32s(*size: int*) → *ValueVec*

Read an array of int32 and return a vec<int32, size>.

read_int64() → int

Read and return a Python int.

read_int64s(*size: int*) → *ValueVec*

Read an array of int64 and return a vec<int64, size>.

read_int8() → int

Read and return a Python int.

read_int8s(*size: int*) → *ValueVec*

Read an array of int8 and return a vec<int8, size>.

read_string() → str

Read and return a Python str.

read_uint16() → int

Read and return a Python int.

read_uint16s(*size: int*) → *ValueVec*

Read an array of uint16 and return a vec<uint16, size>.

read_uint32() → int

Read and return a Python int.

read_uint32s(*size: int*) → *ValueVec*

Read an array of uint32 and return a vec<uint32, size>.

read_uint64() → int

Read and return a Python int.

read_uint64s(*size: int*) → *ValueVec*

Read an array of uint64 and return a vec<uint64, size>.

read_uint8() → int

Read and return a Python int.

read_uint8s(*size: int*) → *ValueVec*

Read an array of uint8 and return a vec<uint8, size>.

read_uuid() → *ValueUUId*

Read and return an uuid.

stream_reading() → *StreamReading*

Return the StreamReading interface.

StreamTokenBinaryWriter

class dsviper.**StreamTokenBinaryWriter**(*stream_raw_reading*)

Bases: object

A class used to write through the StreamRawWriting interface (handle endianness).

stream_writing() → *StreamWriting*

Return the StreamWriting interface.

write_blob(*value:* ValueBlob) → None

Write a blob.

write_blob_id(*value:* ValueBlobId) → None

Write a blob_id.

write_bool(*value: bool*) → None

Write a bool.

write_commit_id(*value:* ValueCommitId) → None

Write a commit_id.

write_double(*value: float*) → None

Write a double.

write_doubles(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of double.

write_float(*value: float*) → None

Write a float.

write_floats(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of float.

write_int16(*value: int*) → None

Write an int16.

write_int16s(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of int16.

write_int32(*value: int*) → None
Write an int32.

write_int32s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of int32.

write_int64(*value: int*) → None
Write an int64.

write_int64s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of int64.

write_int8(*value: int*) → None
Write an int8.

write_int8s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of int8.

write_string(*value: str*) → None
Write a string.

write_uint16(*value: int*) → None
Write an uint16.

write_uint16s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint16.

write_uint32(*value: int*) → None
Write an uint32.

write_uint32s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint32.

write_uint64(*value: int*) → None
Write an uint64.

write_uint64s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint64.

write_uint8(*value: int*) → None
Write an uint8.

write_uint8s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint8.

write_uuid(*value:* ValueUUId) → None
Write an uuid.

Stream I/O

dsviper.StreamReaderBlob	A class used to read data from a blob.
dsviper.StreamReaderFile	A class used to read data from a file.
dsviper.StreamReaderSharedMemory	A class used to read data from a shared memory region.
dsviper.StreamWriterBlob	A class used to write data to a blob.
dsviper.StreamWriterFile	A class used to write data to a file.

continues on next page

Table 28 – continued from previous page

dsviper.StreamWriterSharedMemory	A class used to write data to a shared memory region.

StreamReaderBlob

class dsviper.**StreamReaderBlob**(*blob*)

Bases: object

A class used to read data from a blob.

blob() → *ValueBlob*

Return the blob.

offset() → int

Return the offset.

stream_raw_reading() → *StreamRawReading*

Return the StreamRawReading interface.

StreamReaderFile

class dsviper.**StreamReaderFile**

Bases: object

A class used to read data from a file.

close() → None

Close the file.

offset() → int

Return the offset.

stream_raw_reading() → *StreamRawReading*

Return the StreamRawReading interface.

StreamReaderSharedMemory

class dsviper.**StreamReaderSharedMemory**

Bases: object

A class used to read data from a shared memory region.

offset() → int

Return the offset.

rewind() → None

Rewind the stream.

stream_raw_reading() → *StreamRawReading*

Return the StreamRawReading interface.

StreamWriterBlob

class dsviper.**StreamWriterBlob**

Bases: object

A class used to write data to a blob.

blob() → *ValueBlob*

Return the blob.

stream_raw_writing() → *StreamRawWriting*

Return the StreamRawWriting interface.

StreamWriterFile

class dsviper.**StreamWriterFile**

Bases: object

A class used to write data to a file.

close() → None

Close the file.

offset() → int

Return the offset.

stream_raw_writing() → *StreamRawWriting*

Return the StreamRawWriting interface.

StreamWriterSharedMemory

class dsviper.**StreamWriterSharedMemory**(*shared_memory*)

Bases: object

A class used to write data to a shared memory region.

offset() → int

Return the offset.

rewind() → None

Rewind the stream.

stream_raw_writing() → *StreamRawWriting*

Return StreamRawWriting interface.

Protocols

dsviper.StreamReading	An interface used to read data.
dsviper.StreamWriting	An interface used to read data.
dsviper.StreamEncoding	An interface used to write data to a blob.
dsviper.StreamDecoding	An interface used to read data from a blob.
dsviper.StreamSizing	An interface used to get the size of data.

StreamReading

class dsviper.**StreamReading**

Bases: object

An interface used to read data.

Note: Not directly instantiable.

read_blob() → *ValueBlob*

Read and return a blob.

read_blob_id() → *ValueBlobId*

Read and return a blob_id.

read_bool() → bool

Read and return a Python bool.

read_commit_id() → *ValueCommitId*

Read and return a commit_id.

read_double() → float

Read and return a Python float.

read_doubles(*size: int*) → *ValueVec*

Read an array of doubles and return a vec<double, size>.

read_float() → float

Read aan return a Python float.

read_floats(*size: int*) → *ValueVec*

Read an array of float and return a vec<float, size>.

read_int16() → int

Read and return a Python int.

read_int16s(*size: int*) → *ValueVec*

Read an array of int16 and return a vec<int16, size>.

read_int32() → int

Read and return a Python int.

read_int32s(*size: int*) → *ValueVec*

Read an array of int32 and return a vec<int32, size>.

read_int64() → int

Read and return a Python int.

read_int64s(*size: int*) → *ValueVec*

Read an array of int64 and return a vec<int64, size>.

read_int8() → int

Read and return a Python int.

read_int8s(*size: int*) → *ValueVec*

Read an array of int8 and return a vec<int8, size>.

read_string() → str

Read and return a Python str.

read_uint16() → int

Read and return a Python int.

read_uint16s(*size: int*) → *ValueVec*

Read an array of uint16 and return a vec<uint16, size>.

read_uint32() → int

Read and return a Python int.

read_uint32s(*size: int*) → *ValueVec*

Read an array of uint32 and return a vec<uint32, size>.

read_uint64() → int

Read and return a Python int.

read_uint64s(*size: int*) → *ValueVec*

Read an array of uint64 and return a vec<uint64, size>.

read_uint8() → int

Read and return a Python int.

read_uint8s(*size: int*) → *ValueVec*

Read an array of uint8 and return a vec<uint8, size>.

read_uuid() → *ValueUUId*

Read and return an uuid.

StreamWriting

class dsviper.**StreamWriting**

Bases: object

An interface used to read data.

Note: Not directly instantiable.

write_blob(*value:* ValueBlob) → None

Write a blob.

write_blob_id(*value:* ValueBlobId) → None

Write a blob_id.

write_bool(*value: bool*) → None

Write a bool.

write_commit_id(*value:* ValueCommitId) → None

Write a commit_id.

write_double(*value: float*) → None

Write a double.

write_doubles(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of double.

write_float(*value: float*) → None

Write a float.

write_floats(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of float.

write_int16(*value: int*) → None

Write an int16.

write_int16s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int16.

write_int32(*value: int*) → None
Write an int32.

write_int32s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int32.

write_int64(*value: int*) → None
Write an int64.

write_int64s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int64.

write_int8(*value: int*) → None
Write an int8.

write_int8s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int8.

write_string(*value: str*) → None
Write a string.

write_uint16(*value: int*) → None
Write an uint16.

write_uint16s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint16.

write_uint32(*value: int*) → None
Write an uint32.

write_uint32s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint32.

write_uint64(*value: int*) → None
Write an uint64.

write_uint64s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint64.

write_uint8(*value: int*) → None
Write an uint8.

write_uint8s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint8.

write_uuid(*value:* ValueUUId) → None
Write an uuid.

StreamEncoding

class dsviper.**StreamEncoding**

Bases: `object`

An interface used to write data to a blob.

Note: Not directly instantiable.

end_encoding() → *ValueBlob*

Return the blob and set the flag ended to True.

is_ended() → bool

Return True if the stream is ended.

stream_writing() → *StreamWriting*

Return a StreamWriting interface.

write_blob(*value:* ValueBlob) → None

Write a blob.

write_blob_id(*value:* ValueBlobId) → None

Write a blob_id.

write_bool(*value: bool*) → None

Write a bool.

write_commit_id(*value:* ValueCommitId) → None

Write a commit_id.

write_double(*value: float*) → None

Write a double.

write_doubles(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of double.

write_float(*value: float*) → None

Write a float.

write_floats(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of float.

write_int16(*value: int*) → None

Write an int16.

write_int16s(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of int16.

write_int32(*value: int*) → None

Write an int32.

write_int32s(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of int32.

write_int64(*value: int*) → None

Write an int64.

write_int64s(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of int64.

write_int8(*value: int*) → None

Write an int8.

write_int8s(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of int8.

write_string(*value: str*) → None
Write a string.

write_uint16(*value: int*) → None
Write an uint16.

write_uint16s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint16.

write_uint32(*value: int*) → None
Write an uint32.

write_uint32s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint32.

write_uint64(*value: int*) → None
Write an uint64.

write_uint64s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint64.

write_uint8(*value: int*) → None
Write an uint8.

write_uint8s(*value: Sequence | ValueVec*, *size: int*) → None
Write an array of uint8.

write_uuid(*value: ValueUUId*) → None
Write an uuid.

StreamDecoding

class dsviper.**StreamDecoding**
Bases: object

An interface used to read data from a blob.

Note: Not directly instantiable.

blob() → *ValueBlob*
Return the blob.

has_more() → bool
Return True if there is something to read.

offset() → int
Read the offset in the stream.

read_blob() → *ValueBlob*
Read and return a blob.

read_blob_id() → *ValueBlobId*
Read and return a blob_id.

read_bool() → bool
Read and return a Python bool.

read_commit_id() → *ValueCommitId*

Read and return a commit_id.

read_double() → float

Read and return a Python float.

read_doubles(*size: int*) → *ValueVec*

Read an array of doubles and return a vec<double, size>.

read_float() → float

Read aan return a Python float.

read_floats(*size: int*) → *ValueVec*

Read an array of float and return a vec<float, size>.

read_int16() → int

Read and return a Python int.

read_int16s(*size: int*) → *ValueVec*

Read an array of int16 and return a vec<int16, size>.

read_int32() → int

Read and return a Python int.

read_int32s(*size: int*) → *ValueVec*

Read an array of int32 and return a vec<int32, size>.

read_int64() → int

Read and return a Python int.

read_int64s(*size: int*) → *ValueVec*

Read an array of int64 and return a vec<int64, size>.

read_int8() → int

Read and return a Python int.

read_int8s(*size: int*) → *ValueVec*

Read an array of int8 and return a vec<int8, size>.

read_string() → str

Read a return a Python str.

read_uint16() → int

Read and return a Python int.

read_uint16s(*size: int*) → *ValueVec*

Read an array of uint16 and return a vec<uint16, size>.

read_uint32() → int

Read and return a Python int.

read_uint32s(*size: int*) → *ValueVec*

Read an array of uint32 and return a vec<uint32, size>.

read_uint64() → int

Read and return a Python int.

read_uint64s(*size: int*) → *ValueVec*

Read an array of uint64 and return a vec<uint64, size>.

read_uint8() → int

Read and return a Python int.

read_uint8s(*size: int*) → *ValueVec*

Read an array of uint8 and return a vec<uint8, size>.

read_uuid() → *ValueUUId*

Read and return an uuid.

rewind() → None

Rewind the stream.

stream_reading() → *StreamReading*

Return the StreamReading interface.

StreamSizing

class dsviper.**StreamSizing**

Bases: object

An interface used to get the size of data.

Note: Not directly instantiable.

size_of_blob_id() → int

Return the size of a blob_id.

size_of_bool() → int

Return the size of a bool.

size_of_commit_id() → int

Return the size of a commit_id.

size_of_double() → int

Return the size of a double.

size_of_doubles(*size: int*) → int

Return the size for an array of double.

size_of_float() → int

Return the size of a float.

size_of_floats(*size: int*) → int

Return the size for an array of float.

size_of_int16() → int

Return the size of an int16.

size_of_int16s(*size: int*) → int

Return the size for an array of int16.

size_of_int32() → int

Return the size of an int32.

size_of_int32s(*size: int*) → int
Return the size for an array of int32.

size_of_int64() → int
Return the size of an int64.

size_of_int64s(*size: int*) → int
Return the size for an array of uint64.

size_of_int8() → int
Return the size of an int8.

size_of_int8s(*size: int*) → int
Return the size for an array of int8.

size_of_uint16() → int
Return the size of an uint16.

size_of_uint16s(*size: int*) → int
Return the size for an array of uint16.

size_of_uint32() → int
Return the size of an uint32.

size_of_uint32s(*size: int*) → int
Return the size for an array of uint32.

size_of_uint64() → int
Return the size of an uint64.

size_of_uint64s(*size: int*) → int
Return the size for an array of uint64.

size_of_uint8() → int
Return the size of an uint8.

size_of_uint8s(*size: int*) → int
Return the size for an array of uint8.

size_of_uuid() → int
Return the size of an uuid.

Raw Streams

dsviper.StreamRawReader	An class used to read data through the StreamRawReading interface.
dsviper.StreamRawWriter	A class used to write data through the StreamRawWriting interface.
dsviper.StreamRawReading	An interface used to read data.
dsviper.StreamRawWriting	An interface to write data.

StreamRawReader

class dsviper.**StreamRawReader**(*stream_raw_reading*)

Bases: object

An class used to read data through the StreamRawReading interface.

read_blob() → *ValueBlob*

Read and return a blob.

read_blob_id() → *ValueBlobId*

Read and return a blob_id.

read_bool() → bool

Read and return a Python bool.

read_commit_id() → *ValueCommitId*

Read and return a commit_id.

read_double() → float

Read and return a Python float.

read_doubles(*size: int*) → *ValueVec*

Read an array of doubles and return a vec<double, size>.

read_float() → float

Read aan return a Python float.

read_floats(*size: int*) → *ValueVec*

Read an array of float and return a vec<float, size>.

read_int16() → int

Read and return a Python int.

read_int16s(*size: int*) → *ValueVec*

Read an array of int16 and return a vec<int16, size>.

read_int32() → int

Read and return a Python int.

read_int32s(*size: int*) → *ValueVec*

Read an array of int32 and return a vec<int32, size>.

read_int64() → int

Read and return a Python int.

read_int64s(*size: int*) → *ValueVec*

Read an array of int64 and return a vec<int64, size>.

read_int8() → int

Read and return a Python int.

read_int8s(*size: int*) → *ValueVec*

Read an array of int8 and return a vec<int8, size>.

read_string() → str

Read and return a Python str.

read_uint16() → int

Read and return a Python int.

read_uint16s(*size: int*) → *ValueVec*

Read an array of uint16 and return a vec<uint16, size>.

read_uint32() → int

Read and return a Python int.

read_uint32s(*size: int*) → *ValueVec*

Read an array of uint32 and return a vec<uint32, size>.

read_uint64() → int

Read and return a Python int.

read_uint64s(*size: int*) → *ValueVec*

Read an array of uint64 and return a vec<uint64, size>.

read_uint8() → int

Read and return a Python int.

read_uint8s(*size: int*) → *ValueVec*

Read an array of uint8 and return a vec<uint8, size>.

read_uuid() → *ValueUUId*

Read and return an uuid.

stream_reading() → *StreamReading*

Return the StreamReading interface.

StreamRawWriter

class dsviper.**StreamRawWriter**(*stream_raw_writing*)

Bases: object

A class used to write data through the StreamRawWriting interface.

stream_writing() → *StreamWriting*

Return the StreamWriting interface.

write_blob(*value:* ValueBlob) → None

Write a blob.

write_blob_id(*value:* ValueBlobId) → None

Write a blob_id.

write_bool(*value: bool*) → None

Write a bool.

write_commit_id(*value:* ValueCommitId) → None

Write a commit_id.

write_double(*value: float*) → None

Write a double.

write_doubles(*value: Sequence* | ValueVec, *size: int*) → None

Write an array of double.

write_float(*value: float*) → None

Write a float.

write_floats(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of float.

write_int16(*value: int*) → None
Write an int16.

write_int16s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int16.

write_int32(*value: int*) → None
Write an int32.

write_int32s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int32.

write_int64(*value: int*) → None
Write an int64.

write_int64s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int64.

write_int8(*value: int*) → None
Write an int8.

write_int8s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of int8.

write_string(*value: str*) → None
Write a string.

write_uint16(*value: int*) → None
Write an uint16.

write_uint16s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint16.

write_uint32(*value: int*) → None
Write an uint32.

write_uint32s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint32.

write_uint64(*value: int*) → None
Write an uint64.

write_uint64s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint64.

write_uint8(*value: int*) → None
Write an uint8.

write_uint8s(*value: Sequence* | ValueVec, *size: int*) → None
Write an array of uint8.

write_uuid(*value:* ValueUUId) → None
Write an uuid.

StreamRawReading

class dsviper.**StreamRawReading**

Bases: object

An interface used to read data.

Note: Not directly instantiable.

StreamRawWriting

class dsviper.**StreamRawWriting**

Bases: object

An interface to write data.

Note: Not directly instantiable.

3.3.8 DSM Introspection

DSM classes provide parsing and introspection of Digital Substrate Model files.

When to use: Use `DSMBuilder` to parse `.dsm` files and introspect the resulting definitions (structures, enumerations, attachments, functions).

Quick Start

```
from dsviper import DSMBuilder

# Parse DSM file
builder = DSMBuilder.assemble("model.dsm")
report, dsm_defs, defs = builder.parse()

# Check for parse errors
if report.has_error():
    for err in report.errors():
        print(f"Line {err.line()}: {err.message()}")
    raise RuntimeError("DSM parse failed")

# Introspect structures
for struct in dsm_defs.structures():
    print(f"Struct: {struct.type_name()}")
    for field in struct.fields():
        print(f"  {field.name()}: {field.type_reference()}")

# Introspect attachments
for att in dsm_defs.attachments():
    print(f"Attachment: {att.type_name()}")
    print(f"  Key: {att.key_type()}, Doc: {att.document_type()}")

# Inject constants for runtime use
defs.inject()  # Creates MYAPP_A_*, MYAPP_S_*, etc.
```

Parsing

dsviper.DSMBuilder	A class used to assemble a collection of DSM Definitions from various sources.
dsviper.DSMBuilderPart	A class used to represent a part of the assembled definitions.
dsviper.DSMDefinitions	A class used to represent DSM definitions.
dsviper.DSMDefinitionsInspector	DSMDefinitionsInspector(definitions).
dsviper.DSMParseReport	A class used to collect the error occurred while parsing the assembled definitions.
dsviper.DSMParseError	A class used to represent a parse error.

DSMBuilder

class dsviper.**DSMBuilder**

Bases: object

A class used to assemble a collection of DSM Definitions from various sources.

append(*source: str*, *content: str*) → None

Append a content from a source.

static assemble(*path: str*) → *DSMBuilder*

Return a DSMBuilder with the concatenated content of the DSM files found at the path where the path is a file or a folder.

content() → str

Return the content.

line_offset() → int

Return the line offset.

parse() → tuple[*DSMParseReport*, *DSMDefinitions* | None, *DefinitionsConst* | None]

Parse and return (DSMParseReport, DSMDefinitions, Definitions).

part(*line*) → *DSMBuilderPart* | None

Return the part associated with the line number.

parts() → list[*DSMBuilderPart*]

Return the list of parts.

DSMBuilderPart

class dsviper.**DSMBuilderPart**

Bases: object

A class used to represent a part of the assembled definitions.

Note: Not directly instantiable.

line_end() → int

Return the line end.

line_start() → int

Return the line start.

source() → str

Return the source.

DSMDefinitions

class dsviper.**DSMDefinitions**

Bases: object

A class used to represent DSM definitions. Available representations are plain-text, HTML, JSON, Bson, binary. Use the static factory method to get the definitions from various representation.

Note: Not directly instantiable.

attachment_function_pools() → list[*DSMAttachmentFunctionPool*]

Return the list of attachment function pools.

attachments() → list[*DSMAttachment*]

Return the list of attachments.

static bson_decode(*blob:* ValueBlob) → *DSMDefinitions*

Return a DSMDefinitions from a bson encoded blob.

bson_encode() → *ValueBlob*

Return the definitions in bson.

clubs() → list[*DSMClub*]

Return the list of clubs.

concepts() → list[*DSMConcept*]

Return the list of concepts.

static decode(*blob:* ValueBlob, *, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *DSMDefinitions*

Return the decoded DSMDefinitions with a StreamTokenBinaryCodec if not specified.

encode(*, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *ValueBlob*

Return the blob that encodes the definitions with a StreamTokenBinaryCodec if not specified.

enumerations() → list[*DSMEnumeration*]

Return the list of enumerations.

static from_attachment_function_pool(*attachment_function_pool:* AttachmentFunctionPool, *definitions:* DefinitionsConst) → *DSMDefinitions*

Convert to DSM Definitions.

static from_definitions(*definitions:* DefinitionsConst) → *DSMDefinitions*

Convert to DSM Definitions.

static from_function_pool(*function_pool:* FunctionPool, *definitions:* DefinitionsConst) → *DSMDefinitions*

Convert to DSM Definitions.

function_pools() → list[*DSMFunctionPool*]

Return the list of function pools.

static json_decode(*json_string: str*) → *DSMDefinitions*

Return a DSMDefinitions from a JSON encoded string.

json_encode(*indent: int = -1*) → str

Encode and return the definitions in a JSON encoded string. If indent is specified, the string is pretty printed.

static read(*stream_reading:* StreamReading) → *DSMDefinitions*

Return a DSMDefinitions.

structures() → list[*DSMStructure*]

Return the list of structures.

to_definitions() → *Definitions*

Return a Definitions by converting all DSM definitions.

to_dsm(*, *show_documentation=True*, *show_runtime_id=False*, *html=False*, *attachments: list[*DSMAttachment*] | None = None*) → str

Return the definitions in the DSM language (DSL) sorted by attachment dependencies. All attachments are used, if the parameter attachments is not specified or empty.

write(*stream_writing:* StreamWriting) → None

Write the definitions with the StreamWriting interface.

DSMDefinitionsInspector

class dsviper.**DSMDefinitionsInspector**

Bases: object

DSMDefinitionsInspector(definitions). A class used to retrieve registered concepts, clubs, enumerations, structures from TypeName and attachments from identifier.

attachment_function_pool_ids() → set[*ValueUUId*]

Return the set of uuid for all function pools.

attachment_identifiers() → set[str]

Return the set of identifiers for all attachments.

check_attachment(*identifier: str*) → *DSMAttachment*

Return the attachment or raise.

check_attachment_function_pool(*uuid:* ValueUUId) → *DSMAttachmentFunctionPool*

check_attachmemt_function_pool($self, uuid) –

Return the attachment function pool or raise.

check_club(*type_name:* TypeName) → *DSMClub*

Return the club or raise.

check_concept(*type_name:* TypeName) → *DSMConcept*

Return the concept or raise.

check_enumeration(*type_name:* TypeName) → *DSMEnumeration*

Return the enum or raise.

check_function_pool(*uuid:* ValueUUId) → *DSMFunctionPool*

Return the function pool or raise.

check_structure(*type_name:* TypeName) → *DSMStructure*

Return a struct or raise.

club_type_names() → set[*TypeName*]
Return the set of TypeName for all clubs.

concept_type_names() → set[*TypeName*]
Return the set of TypeName for all concepts.

enumeration_type_names() → set[*TypeName*]
Return the set of TypeName for all enums.

function_pool_ids() → set[*ValueUUId*]
Return the set of uuid for all function pools.

name_spaces() → set[*NameSpace*]
Return the set of NameSpace for all TypNames.

query_attachment(*identifier: str*) → *DSMAttachment* | None
Return the attachment or None.

query_attachment_function_pool(*uuid:* ValueUUId) → *DSMAttachmentFunctionPool* | None
Return the attachment function pool or None.

query_club(*type_name:* TypeName) → *DSMClub* | None
Return the club or None.

query_concept(*type_name:* TypeName) → *DSMConcept* | None
Return the concept or None.

query_enumeration(*type_name:* TypeName) → *DSMEnumeration* | None
Return the enum or None.

query_function_pool(*uuid:* ValueUUId) → *DSMFunctionPool* | None
Return the function pool or None.

query_structure(*type_name:* TypeName) → *DSMStructure* | None
Return a struct or None.

representation(*type_name:* TypeName) → str
Return the shortest representation of the TypeName.

structure_type_names() → set[*TypeName*]
Return the set of TypeName for all structs.

DSMParseReport

class dsviper.**DSMParseReport**
Bases: object

A class used to collect the error occurred while parsing the assembled definitions.

Note: Not directly instantiable.

errors() → list[*DSMParseError*]
Return the list of errors.

has_error() → bool
Return True if errors were occurred.

DSMParseError

class dsviper.**DSMParseError**

Bases: object

A class used to represent a parse error.

Note: Not directly instantiable.

line() → int

Return the line number.

message() → str

Return the message.

pos() → int

Return the position in the line.

source() → str

Return the source.

Model Elements

dsviper.DSMConcept	A class used to represent the definition of a concept.
dsviper.DSMClub	A class used to represent the definition of a club.
dsviper.DSMStructure	A class used to represent the definition of a struct.
dsviper.DSMStructureField	A class used to represent the definition of a field for a struct.
dsviper.DSMEnumeration	A class used to represent the definition of an enum.
dsviper.DSMEnumerationCase	A class used to represent the definition of a case for an enum.
dsviper.DSMAttachment	A class used to represent the definition of an attachment.

DSMConcept

class dsviper.**DSMConcept**

Bases: object

A class used to represent the definition of a concept.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

parent() → *DSMTypeReference*

Return the type reference of the parent concept or None.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_name() → *TypeName*

Return the TypeName.

type_reference() → *DSMTypeReference*

Return the type reference.

DSMClub

class dsviper.**DSMClub**

Bases: object

A class used to represent the definition of a club.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

members() → list[*DSMTypeReference*]

Return the list of members.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_name() → *TypeName*

Return the TypeName.

type_reference() → *DSMTypeReference*

Return the type reference.

DSMStructure

class dsviper.**DSMStructure**

Bases: object

A class used to represent the definition of a struct.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

fields() → list[*DSMStructureField*]

Return the list of fields.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_name() → *TypeName*

Return a TypeName.

type_reference() → *DSMTypeReference*

Return the type reference.

DSMStructureField

class dsviper.**DSMStructureField**

Bases: object

A class used to represent the definition of a field for a struct.

Note: Not directly instantiable.

default_value() → DSMLiteral

Return the default value.

documentation() → str
Return the documentation.

name() → str
Return the name.

type() → DSMType
Return the type.

DSMEnumeration

class dsviper.**DSMEnumeration**
Bases: object

A class used to represent the definition of an enum.

Note: Not directly instantiable.

documentation() → str
Return the documentation.

members() → list[*DSMEnumerationCase*]
Return the list of members.

runtime_id() → *ValueUUId*
Return the uuid assigned by the runtime.

type_name() → *TypeName*
Return the TypeName.

type_reference() → *DSMTypeReference*
Return the type reference.

DSMEnumerationCase

class dsviper.**DSMEnumerationCase**
Bases: object

A class used to represent the definition of a case for an enum.

Note: Not directly instantiable.

documentation() → str
Return the documentation.

name() → str
Return the name.

DSMAttachment

class dsviper.**DSMAttachment**
Bases: object

A class used to represent the definition of an attachment.

Note: Not directly instantiable.

document_type() → DSMType

Return the type reference of the document.

documentation() → str

Return the documentation.

identifier() → str

Return the identifier.

key_type() → *DSMTypeReference*

Return the type reference of the key.

representation() → str

Return the representation.

runtime_id() → *ValueUUId*

Return the uuid assigned by the runtime.

type_name() → *TypeName*

Return the TypeName.

DSM Types

dsviper.DSMTypeKey	A class used to represent the type key<element_type>.
dsviper.DSMTypeVector	A class used to represent the type vector<element_type>.
dsviper.DSMTypeSet	A class used to represent the type set<element_type>.
dsviper.DSMTypeMap	A class used to represent the type map<key_type, element_type>.
dsviper.DSMTypeXArray	A class used to represent the type xarray<element_type>.
dsviper.DSMTypeOptional	A class used to represent the type optional<element_type>.
dsviper.DSMTypeTuple	A class used to represent the type tuple<T0, ...>.
dsviper.DSMTypeVec	A class used to represent the type vec<element_type, size>.
dsviper.DSMTypeMat	A class used to represent the type vec<element_type, columns, rows>.
dsviper.DSMTypeVariant	A class used to represent the type variant<T0, ...>.
dsviper.DSMTypeReference	A class used to represent a reference to a type.

DSMTypeKey

class dsviper.**DSMTypeKey**

Bases: object

A class used to represent the type key<element_type>.

Note: Not directly instantiable.

element_type() → DSMType

Return type the element.

DSMTypeVector

class dsviper.**DSMTypeVector**

Bases: object

A class used to represent the type vector<element_type>.

Note: Not directly instantiable.

element_type() → DSMType

Return type of element.

DSMTypeSet

class dsviper.**DSMTypeSet**

Bases: object

A class used to represent the type set<element_type>.

Note: Not directly instantiable.

element_type() → DSMType

Return the type of element.

DSMTypeMap

class dsviper.**DSMTypeMap**

Bases: object

A class used to represent the type map<key_type, element_type>.

Note: Not directly instantiable.

element_type() → DSMType

Return the type of the element.

key_type() → DSMType

Return the type of the key.

DSMTypeXArray

class dsviper.**DSMTypeXArray**

Bases: object

A class used to represent the type xarray<element_type>.

Note: Not directly instantiable.

element_type() → DSMType

Return type of element.

DSMTypeOptional

class dsviper.**DSMTypeOptional**

Bases: object

A class used to represent the type optional<element_type>.

Note: Not directly instantiable.

element_type() → DSMType
Return the type of the element.

DSMTypeTuple

class dsviper.**DSMTypeTuple**
Bases: object

A class used to represent the type tuple<T0, …>.

Note: Not directly instantiable.

types() → list[DSMType]
Return the list of types.

DSMTypeVec

class dsviper.**DSMTypeVec**
Bases: object

A class used to represent the type vec<element_type, size>.

Note: Not directly instantiable.

element_type() → DSMType
Return the type of element.

size() → int
Return the size.

DSMTypeMat

class dsviper.**DSMTypeMat**
Bases: object

A class used to represent the type vec<element_type, columns, rows>.

Note: Not directly instantiable.

columns() → int
Return the number of columns.

element_type() → DSMType
Return the type of element.

rows() → int
Return the number of rows.

DSMTypeVariant

class dsviper.**DSMTypeVariant**
Bases: object

A class used to represent the type variant<T0, …>.

Note: Not directly instantiable.

types() → list[DSMType]
Return the list of types.

DSMTypeReference

class dsviper.**DSMTypeReference**

Bases: object

A class used to represent a reference to a type.

Note: Not directly instantiable.

domain() → str

Return the domain.

type_name() → *TypeName*

Return a TypeName.

Functions

dsviper.DSMFunction	A class used to represent the definition of a function.
dsviper.DSMFunctionPool	A class used to represent the definition of a function pool.
dsviper.DSMFunctionPrototype	A class used to represent the definition of a function prototype.
dsviper.DSMAttachmentFunction	A class used to represent the definition of an attachment function.
dsviper.DSMAttachmentFunctionPool	A class used to represent the definition of an attachment function pool.

DSMFunction

class dsviper.**DSMFunction**

Bases: object

A class used to represent the definition of a function.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

prototype() → *DSMFunctionPrototype*

Return the prototype.

DSMFunctionPool

class dsviper.**DSMFunctionPool**

Bases: object

A class used to represent the definition of a function pool.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

functions() → list[*DSMFunction*]

Return the list of functions.

name() → str
Return the name.

uuid() → *ValueUUId*
Return the uuid.

DSMFunctionPrototype

class dsviper.**DSMFunctionPrototype**
Bases: object

A class used to represent the definition of a function prototype.

Note: Not directly instantiable.

name() → str
Return the name.

parameters() → list[tuple[str, DSMType]]
Return the list of parameters.

return_type() → DSMType
Return the type of the returned value.

DSMAttachmentFunction

class dsviper.**DSMAttachmentFunction**
Bases: object

A class used to represent the definition of an attachment function.

Note: Not directly instantiable.

documentation() → str
Return the documentation.

is_mutable() → bool
Return True if the function is mutable.

prototype() → *DSMFunctionPrototype*
Return the prototype.

DSMAttachmentFunctionPool

class dsviper.**DSMAttachmentFunctionPool**
Bases: object

A class used to represent the definition of an attachment function pool.

Note: Not directly instantiable.

documentation() → str
Return the documentation.

functions() → list[*DSMAttachmentFunction*]
Return the list of functions.

name() → str
Return the name.

uuid() → *ValueUUId*
Return the uuid.

Literals

dsviper.DSMLiteralValue	A class used to represent the definition of a literal value.
dsviper.DSMLiteralList	A class used to represent the definition of a literal list.

DSMLiteralValue

class dsviper.**DSMLiteralValue**
Bases: object

A class used to represent the definition of a literal value.

Note: Not directly instantiable.

domain() → str
Return the domain.

value() → str
Return the value.

DSMLiteralList

class dsviper.**DSMLiteralList**
Bases: object

A class used to represent the definition of a literal list.

Note: Not directly instantiable.

members() → list[DSMLiteral]
Return the list of members.

3.3.9 HTML Rendering

Classes for rendering Viper data as HTML in web and desktop applications.

When to use: Use `DocumentNode` to traverse nested Viper documents with full metadata. Use `Html` to generate styled HTML output for Flask web apps or Qt desktop apps (via `QTextEdit.setHtml()`).

Quick Start

```
from flask import Flask, render_template
from dsviper import CommitDatabase, DocumentNode, Html, ValueKey

app = Flask(__name__)

@app.get("/document/<uuid:instance_id>")
def document_view(instance_id):
    db = CommitDatabase.open("model.cdb")
```

(continues on next page)

(continued from previous page)

```
    db.definitions().inject()

    # Create key and get document accessor
    key = ValueKey.create(MYAPP_C_Entity, str(instance_id))
    attachment_getting = db.state(db.last_commit_id()).attachment_getting()

    # Build document tree for all attachments
    nodes = DocumentNode.create_documents(key, attachment_getting)

    # Render as collapsible HTML details
    content = Html.documents_details(nodes, show_type=True)
    return render_template("document.html", content=content)
```

DocumentNode Tree

`DocumentNode` provides a recursive tree structure for navigating Viper documents:

```
from dsviper import DocumentNode

# Create tree from a key
nodes = DocumentNode.create_documents(key, attachment_getting)

for node in nodes:
    print(f"{node.string_component()}: {node.string_value()}")

    # Type introspection for conditional rendering
    if node.is_editable():
        if node.is_boolean():
            render_checkbox(node)
        elif node.is_string():
            render_text_input(node)
        elif node.is_enumeration():
            render_select(node)

    # Recursive traversal
    if node.is_expandable():
        for child in node.children():
            render_node(child)
```

Custom HTML Rendering

Build custom renderers using `DocumentNode` metadata:

```
from dsviper import DocumentNode, ValueKey, ValueEnumeration

def render_node(node: DocumentNode, level: int = 0) -> str:
    indent = "  " * level
    html = ""

    if node.is_expandable():
        # Collapsible container
        html += f'{indent}<details open>'
```

(continues on next page)

(continued from previous page)

```
        html += f'<summary>{node.string_component()}</summary>'
        for child in node.children():
            html += render_node(child, level + 1)
        html += f'{indent}</details>'
    else:
        # Leaf value
        html += f'{indent}<div>{node.string_component()} = '

        if node.is_editable():
            # Editable field with form input
            html += f'<input name="{node.uuid().encoded()}" '
            html += f'value="{node.string_value()}"/>'
        else:
            html += node.string_value()

        html += '</div>'

    return html
```

Node metadata available:

Method	Description
`path()`	Path to this node (for `update()` mutations)
`key()`	Document key (for identifying the document)
`attachment()`	Attachment containing this document
`uuid()`	Unique ID for this node (useful for HTML element IDs)
`is_editable()`	Whether this field can be modified
`is_expandable()`	Whether this node has children
`children()`	Child nodes for containers/structures

Html Helpers

The `Html` class provides ready-to-use rendering:

```
from dsviper import Html, DocumentNode

# Render documents as collapsible details
nodes = DocumentNode.create_documents(key, attachment_getting)
content = Html.documents_details(nodes, show_type=True)

# Render a single value with syntax highlighting
html = Html.value(my_structure)

# Pretty-print with indentation
html = Html.value_pretty(my_structure, show_type=True)

# Render DSM schema as HTML
html = Html.dsm_definitions(dsm_defs, show_documentation=True)

# Build complete HTML document
page = Html.document(
```

(continues on next page)

(continued from previous page)

```
    title="My Document",
    style=Html.style(),
    body=Html.body(content)
)
```

Qt Desktop Integration

The same `Html` helpers work in Qt/PySide6 applications via `QTextEdit`:

```
from PySide6.QtWidgets import QTextEdit
from dsviper import Html, CommitDatabase

# In a Qt dialog or widget
class InspectDialog:
    def __init__(self):
        self.text_edit = QTextEdit()

    def show_definitions(self, db: CommitDatabase):
        # Get DSM definitions with HTML formatting
        dsm_defs = db.definitions().to_dsm_definitions()
        content = Html.dsm_definitions(dsm_defs, show_documentation=True)

        # Build complete HTML document
        style = Html.style()
        body = Html.body(content)
        document = Html.document("DSM Definitions", style, body)

        # Display in QTextEdit
        self.text_edit.setHtml(document)

    def show_value(self, value):
        content = Html.value(value, use_description=True)
        document = Html.document("Value", Html.style(), Html.body(content))
        self.text_edit.setHtml(document)
```

This pattern is used by `cdbe.py` (CDB Editor) and `dbe.py` (Database Editor) to display definitions and values with syntax highlighting.

Choosing the Right Approach

Use Case	API	Note
Quick document display	`Html.documents_details()`	Ready-to-use collapsible view
Custom form builder	`DocumentNode` traversal	Full control over rendering
Value debugging	`Html.value_pretty()`	Syntax-highlighted output
Schema documentation	`Html.dsm_definitions()`	Display DSM structure
Qt desktop display	`Html.document()` + `setHtml()`	Works with QTextEdit

DocumentNode

dsviper.DocumentNode	A class used to represent a node in the tree representation of a value.

DocumentNode

class dsviper.**DocumentNode**

Bases: object

A class used to represent a node in the tree representation of a value.

Note: Not directly instantiable.

attachment() → *Attachment*

Return the attachment.

children() → list[*DocumentNode*]

Return the list of child.

static create_documents(*key:* ValueKey, *attachment_getting:* AttachmentGetting) → list[*DocumentNode*]

Return a list of documents as a hierarchy of DocumentNode.

document() → *Value*

Return the document.

is_blob_id() → bool

Return True the node is a blob_id.

is_boolean() → bool

Return True the node is a boolean.

is_collection() → bool

Return True the node is a collection.

is_container() → bool

Return True it's a container.

is_double() → bool

Return True the node is a double.

is_editable() → bool

Return True if it's editable.

is_enumeration() → bool

Return True the node is an enumeration.

is_expandable() → bool

Return True the node is expandable.

is_float() → bool

Return True the node is a float.

is_int16() → bool

Return True the node is an int16.

is_int32() → bool
Return True the node is an int32.

is_int64() → bool
Return True the node is an int64.

is_int8() → bool
Return True the node is an int8.

is_integer() → bool
Return True the node is an integer.

is_key() → bool
Return True the node is a key.

is_numeric() → bool
Return True the node is a number.

is_primitive() → bool
Return True the node is a primitive.

is_readonly() → bool
Return True if it's readonly.

is_real() → bool
Return True the node is real.

is_string() → bool
Return True the node is a string.

is_uint16() → bool
Return True the node is an uint16.

is_uint32() → bool
Return True the node is an uint32.

is_uint64() → bool
Return True the node is an uint64.

is_uint8() → bool
Return True the node is an uint8.

is_uuid() → bool
Return True the node is an uuid.

key() → *ValueKey*
Return the key.

parent() → *DocumentNode* | None
Return the parent or None.

path() → *PathConst*
Return the path.

string_component() → str
Return the string component.

string_component_tooltip() → str
Return the string component tooltip.

string_path() → str
Return the string path.

string_type() → str
Return the string type.

string_value() → str
Return the string_value.

string_value_tooltip() → str
Return the string value tooltip.

type() → str
Return the type of node.

uuid() → *ValueUUId*
Return the uuid.

value() → *Value*
Return the value.

Html

dsviper.Html	A class used to generate HTML representation.

Html

class dsviper.**Html**
Bases: object

A class used to generate HTML representation.

Note: Not directly instantiable.

static body(*content: str*) → str
Embed the content in a body.

static document(*title: str*, *style: str*, *body: str*) → str
Return an HTML document.

static documents_details(*documents: list[*DocumentNode*]*, *show_type: bool = False*) → str
Return the HTML representation of the documents as a hierarchy of details.

static dsm_definitions(*definitions:* DSMDefinitions, *show_documentation: bool = False*, *show_runtime_id: bool = False*) → str
Return the HTML representation of the DSM Definitions.

static style() → str
Return the default style.

static type(*type:* Type) → str
Return the representation of the type in HTML.

static value(*value:* Value, *use_description: bool = False*) → str

Return the representation of the value in HTML.

static value_pretty(*value:* Value, *show_type: bool = False*) → str

Return the representation of the value like the HTML pretty printer.

3.3.10 Core Utilities

Core classes for definitions, namespaces, paths, and utilities.

When to use: Use `Definitions` to register custom types, `Path` to navigate nested structures, and `Logging` for debug output.

Quick Start

```
from dsviper import Definitions, NameSpace, ValueUUId, Path, LoggerConsole, Logging

# Create namespace for your types
ns = NameSpace(ValueUUId("f529bc42-0618-4f54-a3fb-d55f95c5ad03"), "MyApp")

# Create and register definitions
defs = Definitions()

# Paths navigate nested structures
path = Path.from_field("user").field("address").field("city").const()
city = path.at(document)  # Get value at path
path.set(document, "Paris")  # Set value at path

# Logging for debugging
logger = LoggerConsole(Logging.LEVEL_DEBUG)
log = logger.logging()
log.info("Application started")
log.error("Something went wrong")
```

Key Classes

Class	Purpose	Example
`Definitions`	Register custom types	`defs = Definitions()`
`NameSpace`	Group types by namespace	`ns = NameSpace(uuid, "App")`
`Path`	Navigate nested data	`Path.from_field("x").field("y")`
`Logging`	Debug output	`logger.logging().info(msg)`
`Error`	Parse error messages	`Error.parse(str(e))`

Definitions

dsviper.Definitions	A class used to register concept, club, enumeration, structure and attachment.
dsviper.DefinitionsConst	A class used to retreive registered concepts, clubs, enumerations, structures and attachments.
dsviper.DefinitionsCollector	A class used to collect referenced types.

continues on next page

Table 38 – continued from previous page

dsviper.DefinitionsInspector	DefinitionsInspector(definitions).
dsviper.DefinitionsMapper	A class use for INTERNAL DEVELOPMENT.
dsviper.DefinitionsExtendInfo	A class used to represent the types exchanged during the synchronization of two databases.

Definitions

class dsviper.**Definitions**

Bases: `object`

A class used to register concept, club, enumeration, structure and attachment.

const() → *DefinitionsConst*

Return the DefinitionsConst interface.

create_attachment(*namespace:* NameSpace, *name: str*, *key_type: _AbstractionTypes*, *document_type:* Type, *, *documentation: str | None = None*) → *Attachment*

Create an attachment.

create_club(*namespace:* NameSpace, *name: str*, *, *documentation: str | None = None*) → *TypeClub*

Create and return a club.

create_concept(*namespace:* NameSpace, *name: str*, *, *documentation: str | None = None*, *parent:* TypeConcept | *None = None*) → *TypeConcept*

Create and return a concept.

create_enumeration(*namespace:* NameSpace, *enumeration_descriptor:* TypeEnumerationDescriptor) → *TypeEnumeration*

Create and return an enum from a descriptor.

create_membership(*type_club:* TypeClub, *type_concept:* TypeConcept) → None

Create a club membership.

create_structure(*namespace:* NameSpace, *structure_descriptor:* TypeStructureDescriptor) → *TypeStructure*

Create and return a struct from a descriptor.

static decode(*blob:* ValueBlob, *, *stream_codec_instancing:* StreamCodecInstancing | *None = None*) → *Definitions*

Return a Definitions by decoding the blob with a StreamTokenBinaryCodec if not specified.

extend(*definitions:* DefinitionsConst) → *DefinitionsExtendInfo*

Extend the registered types with the types of definitions and return information about added definitions.

extend_concepts(*definitions:* DefinitionsConst) → set[*ValueUUId*]

Extend the registered concepts with the concepts of definitions.

static read(*stream_reading:* StreamReading) → *Definitions*

Read and return a Definitions.

DefinitionsConst

class dsviper.**DefinitionsConst**

Bases: object

A class used to retreive registered concepts, clubs, enumerations, structures and attachments.

Note: Not directly instantiable.

attachment_runtime_ids() → set[*ValueUUId*]

Return the set of runtime ID for all attachments.

attachments() → list[*Attachment*]

Return the list of attachments.

check_attachment(*attachment_runtime_id:* ValueUUId) → *Attachment*

Return the attachment or raise.

check_club(*runtime_id:* ValueUUId) → *TypeClub*

Return the club or raise.

check_concept(*runtime_id:* ValueUUId) → *TypeConcept*

Return the concept or raise.

check_enumeration(*runtime_id:* ValueUUId) → *TypeEnumeration*

Return the enum or raise.

check_structure(*runtime_id:* ValueUUId) → *TypeStructure*

Return a struct or raise.

check_type(*runtime_id:* ValueUUId) → *Type*

Return the type or raise.

club_runtime_ids() → set[*ValueUUId*]

Return the set of runtime ID for all clubs.

clubs() → list[*TypeClub*]

Return the list of clubs.

collector() → *DefinitionsCollector*

Return a collector.

concept_members(*type_concept:* TypeConcept) → list[*TypeConcept*]

Return the list of related concepts.

concept_runtime_ids() → set[*ValueUUId*]

Return the set of runtime ID for all concepts.

concepts() → list[*TypeConcept*]

Return the list of concepts.

contains(*definitions:* DefinitionsConst) → bool

return True if the types and attachments of the definitions are known.

copy() → *Definitions*

Return a copy immutable of the definitions.

discard(*namespace: dict | None = None*) → None

Remove injected constants from a namespace. If namespace is None, the __main__ module is used.

encode(*, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *ValueBlob*

Return the blob that encodes the definitions with a StreamTokenBinaryCodec if not specified.

enumeration_runtime_ids() → set[*ValueUUId*]

Return the set of runtime ID for all enums.

enumerations() → list[*TypeEnumeration*]

Return the list of enums.

extract(*definitions_collector:* DefinitionsCollector) → *Definitions*

Create and return a new Definitions from the collected types.

hexdigest() → str

Return the hexdigest.

inject(*namespace: dict* | *None* = *None*) → None

Fill a namespace with constants for all definitions. If namespace is None, the __main__ module is used. When using an embedded editor, pass globals() as namespace. <namespace>_T_<name> for a concept or a club. <namespace>_K_<name> for a key of a concept or a club. <namespace>_S_<name> for a structure. <namespace>_E_<name> for a enumeration. <namespace>_P_<name>_… is the path for a property of the structure. <namespace>_A_<key.name>_<name> for an attachment.

is_equal(*definitions:* DefinitionsConst) → bool

Return True if self and definitions are equals.

query_attachment(*attachment_runtime_id:* ValueUUId) → *Attachment* | None

Return the attachment or None.

query_club(*runtime_id:* ValueUUId) → *TypeClub* | None

Return the club or None.

query_concept(*runtime_id:* ValueUUId) → *TypeConcept* | None

Return the concept or None.

query_enumeration(*runtime_id:* ValueUUId) → *TypeEnumeration* | None

Return the enum or None.

query_structure(*runtime_id:* ValueUUId) → *TypeStructure* | None

Return a struct or None.

query_type(*runtime_id:* ValueUUId) → *Type* | None

Return the type or None.

query_types(*name: str*) → list[*Type*]

Return the list of types.

runtime_ids() → set[*ValueUUId*]

Return the set of runtime ID for all registered types.

structure_runtime_ids() → set[*ValueUUId*]

Return the set of runtime ID for all structs.

structures() → list[*TypeStructure*]

Return the list of structs.

to_dsm_definitions() → *DSMDefinitions*

Convert and return a DSMDefinition.

types() → list[*Type*]

Return the topologically sorted list of types.

write(*stream_writing:* StreamWriting) → None

Write to a stream.

DefinitionsCollector

class dsviper.**DefinitionsCollector**

Bases: object

A class used to collect referenced types.

Note: Not directly instantiable.

club_runtime_ids() → set[*ValueUUId*]

Return the set of uuid for collected clubs.

collect_attachment(*attachment:* Attachment) → None

Collect the types referenced by the attachment.

collect_prototype(*prototype:* FunctionPrototype) → None

Collect the types referenced by the prototype.

collect_prototypes(*prototypes: list[*FunctionPrototype*]*) → None

Collect the types referenced by the list of prototypes.

collect_structure_descriptor(*type_structure_descriptor:* TypeStructureDescriptor) → None

Collect the types referenced by a structure descriptor.

collect_type(*type:* Type) → None

Collect the types referenced by the type.

concept_runtime_ids() → set[*ValueUUId*]

Return the set of uuid for collected concepts.

enumeration_runtime_ids() → set[*ValueUUId*]

Return the set of uuid for collected enumerations.

has_any() → bool

Return True if the type any was collected.

has_any_concept() → bool

Return True if the type any_concept was collected.

structure_runtime_ids() → set[*ValueUUId*]

Return the set of uuid for collected structures.

DefinitionsInspector

class dsviper.**DefinitionsInspector**

Bases: object

DefinitionsInspector(definitions). A class used to retrieve registered concepts, clubs, enumerations, structures and attachments from TypeName.

attachment_identifiers() → set[str]

Return the set of identifiers for all attachments.

check_attachment(*identifier: str*) → *Attachment*

Return the attachment or raise.

check_club(*type_name:* TypeName) → *TypeClub*

Return the club or raise.

check_concept(*type_name:* TypeName) → *TypeConcept*

Return the concept or raise.

check_enumeration(*type_name:* TypeName) → *TypeEnumeration*

Return the enum or raise.

check_structure(*type_name:* TypeName) → *TypeStructure*

Return a struct or raise.

club_type_names() → set[*TypeName*]

Return the set of TypeName for all clubs.

concept_type_names() → set[*TypeName*]

Return the set of TypeName for all concepts.

enumeration_type_names() → set[*TypeName*]

Return the set of TypeName for all enums.

name_spaces() → set[*NameSpace*]

Return the set of NameSpace for all TypNames.

query_attachment(*identifier: str*) → *Attachment* | None

Return the attachment or None.

query_club(*type_name:* TypeName) → *TypeClub* | None

Return the club or None.

query_concept(*type_name:* TypeName) → *TypeConcept* | None

Return the concept or None.

query_enumeration(*type_name:* TypeName) → *TypeEnumeration* | None

Return the enum or None.

query_structure(*type_name:* TypeName) → *TypeStructure* | None

Return a struct or None.

representation(*type_name:* TypeName) → str

Return the shortest representation of the TypeName.

structure_type_names() → set[*TypeName*]

Return the set of TypeName for all structs.

DefinitionsMapper

class dsviper.**DefinitionsMapper**(*source_definitions*, *target_definitions*)

Bases: object

A class use for INTERNAL DEVELOPMENT.

attachment(*attachment:* Attachment) → *Attachment*

Return the mapped attachment.

source() → *DefinitionsConst*
Return the source.

target() → *DefinitionsConst*
Return the target.

type(*type:* Type) → *Type*
Return the mapped type.

value(*value:* Value) → *Value*
Return the mapped value.

DefinitionsExtendInfo

class dsviper.**DefinitionsExtendInfo**
Bases: object

A class used to represent the types exchanged during the synchronization of two databases.

Note: Not directly instantiable.

attachment_runtime_ids() → set[*ValueUUId*]
Return the set of runtime ID for all added attachments.

club_runtime_ids() → set[*ValueUUId*]
Return the set of runtime ID for all added clubs.

concept_runtime_ids() → set[*ValueUUId*]
Return the set of runtime ID for all added concepts.

count() → int
Return the count of added definitions.

enumeration_runtime_ids() → set[*ValueUUId*]
Return the set of runtime ID for all added enumerations.

memberships() → dict[*ValueUUId*, *ValueUUId*]
Return a dict[uuid, set[uuid]] for all memberships.

structure_runtime_ids() → set[*ValueUUId*]
Return the set of runtime ID for all structures.

Namespace

dsviper.NameSpace	A class used to describe a namespace.

NameSpace

class dsviper.**NameSpace**(*uuid*, *name*)
Bases: object

A class used to describe a namespace.

GLOBAL = ::

name() → str

Return the name.

uuid() → *ValueUUId*

Return the uuid.

Paths

dsviper.Path	A class used to construct the location of a portion of a value.
dsviper.PathConst	A class used to retreive a value from a Path.
dsviper.PathComponent	A class used to represent a component of a Path.
dsviper.PathElementInfo	A class used to represent the various information for an element of a set.
dsviper.PathEntryKeyInfo	A class used to represent various information for an entry of a map.

Path

class dsviper.**Path**(*path=None*)

Bases: object

A class used to construct the location of a portion of a value.

Use the static factory method from_field(…), from_index(…), from_key(…), from_position(…) and from_unwrap().

const() → *PathConst*

Return the PathConst interface.

copy() → *Path*

Return a copy.

static decode(*blob:* ValueBlob, *definitions:* DefinitionsConst, *, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *Path*

Return a Path by decoding the blob with a StreamBinaryCodec if not specified.

element(*index: int*) → *Path*

Append an element component and return self.

entry(*key: _InputValues*) → *Path*

Append an entry component and return self.

field(*name: str*) → *Path*

Append a field component and return self.

static from_element(*index: int*) → *Path*

Create and return a Path with an element component at index.

static from_entry(*value: _InputValues*) → *Path*

Create and return a Path with an entry component.

static from_field(*name: str*) → *Path*

Create and return a Path with a field component.

static from_index(*index: int*) → *Path*

Create a return a Path with an index component.

static from_key(*key: _InputValues*) → *Path*

Create and return a Path with a key component.

static from_position(*position:* ValueUUId) → *Path*

Create and return a Path with a position component.

static from_unwrap() → *Path*

Create and return a Path with an unwrap component.

index(*value: int*) → *Path*

Append an index component and return self.

key(*value: _InputValues*) → *Path*

Append a key component and return self.

path(*other:* PathConst) → *Path*

Append path components and return self.

position(*value:* ValueUUId) → *Path*

Append a position component and return self.

static read(*stream_reading:* StreamReading, *definitions:* DefinitionsConst) → *Path*

Read and return a Path.

representation() → str

Return a string representation.

unwrap() → *Path*

Append an unwrap directive component and return self.

PathConst

class dsviper.**PathConst**

Bases: object

A class used to retreive a value from a Path.

Note: Not directly instantiable.

ancestors() → list[*Path*]

Return the list of ancestors.

at(*target:* Value, *, *encoded: bool = True*) → _OutputValues

Return the value for the target.

check_type(*type:* Type) → *Type*

Check if the path can be used to locate the value in the type.

components() → list[*PathComponent*]

Return the list of components.

copy() → *Path*

Return a copy.

element_info() → *PathElementInfo*

Return the ElementInfo.

encode(*, *stream_codec_instancing:* StreamCodecInstancing | *None* = *None*) → *ValueBlob*

Return a blob that encodes the Path with a StreamTokenBinaryCodec if not specified.

entry_key_info() → *PathEntryKeyInfo*

Return the EntryKeyInfo.

from_component(*index: int*) → *Path*

Return a path from index.

has_prefix(*path:* PathConst) → bool

Return True if other is a prefix.

is_applicable(*value:* Value) → bool

Check if the path is applicable to the value.

is_element_path() → bool

Return True if the path contains Element components.

is_entry_key_path() → bool

Return True if the path contains Entry components.

is_regular() → bool

Return False if the path contains Entry components.

is_root() → bool

Return True if self is the root.

last_component() → *Path*

Return the last component as a Path.

last_component_value(*, *encoded: bool* = *True*) → _OutputValues | None

Return the last component value.

parent() → *Path*

Return the parent Path or raise if self is root.

patch(*target:* Value, *value: _InputValues*) → None

Assign the value to the target.

regularized() → *Path*

Return the path without the entry component.

representation() → str

Return a string representation.

set(*target:* Value, *value: _InputValues*) → None

Assign the value to the target.

to_component(*index: int*) → *Path*

Return a path to index (excluded).

write(*, *stream_writing:* StreamWriting) → None

Write a path.

PathComponent

class dsviper.**PathComponent**

Bases: object

A class used to represent a component of a Path.

Note: Not directly instantiable.

type() → str

Return the type.

value(*, *encoded: bool = True*) → _OutputValues | None

Return the value or None.

PathElementInfo

class dsviper.**PathElementInfo**

Bases: object

A class used to represent the various information for an element of a set.

Note: Not directly instantiable.

element_path() → *PathConst*

Return the path for the element.

index() → int

Return the index for the element.

set_path() → *PathConst*

Return the path for the set.

PathEntryKeyInfo

class dsviper.**PathEntryKeyInfo**

Bases: object

A class used to represent various information for an entry of a map.

Note: Not directly instantiable.

key() → *Value*

Return the key.

key_path() → *PathConst*

Return the path for the key.

map_path() → *PathConst*

Return the path for the map.

Function Pools

dsviper.Function	A class used to call a C++ function.
dsviper.FunctionPool	A class used to register and retreive C++ functions.
dsviper.FunctionPoolFunctions	A class used to represent the functions of a pool.

continues on next page

Table 41 – continued from previous page

dsviper.FunctionPrototype	A class used to represent the prototype of a function.

Function

class dsviper.**Function**

Bases: object

A class used to call a C++ function.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

prototype() → *FunctionPrototype*

Return the prototype.

FunctionPool

class dsviper.**FunctionPool**

Bases: object

A class used to register and retreive C++ functions.

Note: Not directly instantiable.

check(*func_name: str*) → *Function*

Return the function or raise.

documentation() → str

Return the documentation.

funcs → *FunctionPoolFunctions*

all functions.

name() → str

Return the name.

query(*func_name: str*) → *Function* | None

Return the function or None.

uuid() → *ValueUUId*

Return the uuid.

FunctionPoolFunctions

class dsviper.**FunctionPoolFunctions**

Bases: object

A class used to represent the functions of a pool.

Note: Not directly instantiable.

FunctionPrototype

class dsviper.**FunctionPrototype**

Bases: object

A class used to represent the prototype of a function.

Note: Not directly instantiable.

parameters() → list[tuple[str, *Type*]]

Return the list of parameters.

return_type() → *Type*

Return the type of the returned value.

Hashing

dsviper.Hashing	An interface to abstract a hasher.
dsviper.HashCRC32	A class used to hash data with CRC32.
dsviper.HashMD5	A class used to hash data with MD5.
dsviper.HashSHA1	A class used to hash data with SHA1.
dsviper.HashSHA256	A class used to hash data with SHA256.
dsviper.HashSHA3	A class used to hash a data with SHA3.

Hashing

class dsviper.**Hashing**

Bases: object

An interface to abstract a hasher.

Note: Not directly instantiable.

block_size() → int

Return the internal block size of the hash algorithm in bytes.

digest() → *ValueBlob*

Return the digest.

digest_size() → int

Return the size of the resulting hash in bytes.

hexdigest() → str

Return the digest in hexadecimal.

name() → str

Return the name.

reset() → None

Reset the hasher.

update(*blob:* ValueBlob) → None

Update the data.

HashCRC32

class dsviper.**HashCRC32**

Bases: object

A class used to hash data with CRC32.

block_size() → int

Return the internal block size of the hash algorithm in bytes.

digest() → *ValueBlob*

Return the digest.

digest_size() → int

Return the size of the resulting hash in bytes.

static hash(*blob:* ValueBlob) → str

Return the hexdigest of the blob.

hashing() → *Hashing*

Return the Hashing interface.

hexdigest() → str

Return the digest in hexadecimal.

name() → str

Return the name.

reset() → None

Reset the hasher.

update(*blob:* ValueBlob) → None

Update the data.

HashMD5

class dsviper.**HashMD5**

Bases: object

A class used to hash data with MD5.

block_size() → int

Return the internal block size of the hash algorithm in bytes.

digest() → *ValueBlob*

Return the digest.

digest_size() → int

Return the size of the resulting hash in bytes.

static hash(*blob:* ValueBlob) → str

Return the hexdigest of the blob.

hashing() → *Hashing*

Return the Hashing interface.

hexdigest() → str

Return the digest in hexadecimal.

name() → str
Return the name.

reset() → None
Reset the hasher.

update(*blob:* ValueBlob) → None
Update the data.

HashSHA1

class dsviper.**HashSHA1**
Bases: object

A class used to hash data with SHA1.

block_size() → int
Return the internal block size of the hash algorithm in bytes.

digest() → *ValueBlob*
Return the digest.

digest_size() → int
Return the size of the resulting hash in bytes.

static hash(*blob:* ValueBlob) → str
Return the hexdigest of the blob.

hashing() → *Hashing*
Return the Hashing interface.

hexdigest() → str
Return the digest in hexadecimal.

name() → str
Return the name.

reset() → None
Reset the hasher.

update(*blob:* ValueBlob) → None
Update the data.

HashSHA256

class dsviper.**HashSHA256**
Bases: object

A class used to hash data with SHA256.

block_size() → int
Return the internal block size of the hash algorithm in bytes.

digest() → *ValueBlob*
Return the digest.

digest_size() → int
Return the size of the resulting hash in bytes.

static hash(*blob:* ValueBlob) → str
Return the hexdigest of the blob.

hashing() → *Hashing*
Return the Hashing interface.

hexdigest() → str
Return the digest in hexadecimal.

name() → str
Return the name.

reset() → None
Reset the hasher.

update(*blob:* ValueBlob) → None
Update the data.

HashSHA3

class dsviper.**HashSHA3**(*bits*)
Bases: object

A class used to hash a data with SHA3. Supported bits are 224, 256, 384 or 512.

block_size() → int
Return the internal block size of the hash algorithm in bytes.

digest() → *ValueBlob*
Return the digest.

digest_size() → int
Return the size of the resulting hash in bytes.

static hash(*blob:* ValueBlob, *bits: str = '256'*) → str
Return the hexdigest of the blob.

hashing() → *Hashing*
Return the Hashing interface.

hexdigest() → str
Return the hexa digest.

name() → str
Return the name.

reset() → None
Reset the hasher.

update(*blob:* ValueBlob) → None
Update the data.

Logging

dsviper.Logging	An interface to emit a message.
dsviper.LoggerConsole	A class used to display a message on the console.

continues on next page

Table 43 – continued from previous page

dsviper.LoggerPrint	A class used to display a message on the Python stdout.
dsviper.LoggerReport	A class used to collect messages.
dsviper.LoggerNull	A class used to discard messages.

Logging

class dsviper.**Logging**

Bases: object

An interface to emit a message.

Note: Not directly instantiable.

LEVEL_ALL = 0

LEVEL_CRITICAL = 50

LEVEL_DEBUG = 10

LEVEL_ERROR = 40

LEVEL_INFO = 20

LEVEL_WARNING = 30

static create(*object*) → *Logging*

Return a new Logging if the Python object responds to the interface or raise.

critical(*message: str*) → None

Log the message for level Critical.

debug(*message: str*) → None

Log the message for level Debug.

error(*message: str*) → None

Log the message for level Error.

info(*message: str*) → None

Log the message for level Info.

log(*level: int*, *message: str*) → None

Log the message for level.

warning(*message: str*) → None

Log the message for level Warning.

LoggerConsole

class dsviper.**LoggerConsole**(*level=0*)

Bases: object

A class used to display a message on the console.

logging() → *Logging*

Return the Logging interface.

LoggerPrint

class dsviper.**LoggerPrint**(*level=0*)

Bases: object

A class used to display a message on the Python stdout.

logging() → *Logging*

Return the Logging interface.

LoggerReport

class dsviper.**LoggerReport**(*level=0*)

Bases: object

A class used to collect messages.

logging() → *Logging*

Return the Logging interface.

messages() → list[str]

Return the list of messages.

LoggerNull

class dsviper.**LoggerNull**(*level=0*)

Bases: object

A class used to discard messages.

logging() → *Logging*

Return the Logging interface.

Remote Services

dsviper.ServiceRemote	A class used to connect to a remote service.
dsviper.ServiceRemoteFunction	A class used to call a remote function.
dsviper.ServiceRemoteFunctionPool	A class used to represent a remote function pool.
dsviper.ServiceRemoteFunctionPoolFunction	A class used to represent a remote function.
dsviper.ServiceRemoteFunctionPoolFunctions	A class used to represent the functions of a pool.
dsviper.ServiceRemoteFunctionPools	A class used to represent the remote function pools.
dsviper.ServiceRemoteAttachmentFunction	A class used to call a remote attachment function.
dsviper.ServiceRemoteAttachmentFunctionPoo.	A class used to represent a remote attachment function pool.
dsviper.ServiceRemoteAttachmentFunctionPoo.	A class used to represent a remote attachment function.
dsviper.ServiceRemoteAttachmentFunctionPoo.	A class used to represent the remote attachment functions.
dsviper.ServiceRemoteAttachmentFunctionPoo.	A class used to represent the remote attachment pools.

ServiceRemote

class dsviper.**ServiceRemote**

Bases: object

A class used to connect to a remote service. Use the static factory method connect(…) or connect_local(…).

Note: Not directly instantiable.

attachment_function_pool_funcs(*pool_id_or_name:* ValueUUId | *str*) → *ServiceRemoteAttachmentFunctionPoolFunctions* | None

Return functions or None.

attachment_function_pools() → list[*ServiceRemoteAttachmentFunctionPool*]

Return a list of pools.

attachment_pools → *ServiceRemoteAttachmentFunctionPools*

all attachment function pools.

close() → None

Close the service.

static connect(*host: str*, *service: str*, *definitions:* Definitions) → *ServiceRemote*

Connect to a remote service.

static connect_local(*socket_path: str*, *definitions:* Definitions) → *ServiceRemote*

Connect to a service located at socket_path.

definitions() → *DefinitionsConst*

Return the definitions.

function_pool_funcs(*pool_id_or_name:* ValueUUId | *str*) → *ServiceRemoteFunctionPoolFunctions* | None

Return functions or None.

function_pools() → list[*ServiceRemoteFunctionPool*]

Return a list of pools.

is_closed() → bool

Return True if the service is closed.

peername() → str

Return the peername.

pools → *ServiceRemoteFunctionPools*

all function pools.

sockname() → str

Return the sockname.

to_dsm_definitions() → *DSMDefinitions*

Return the DSMDefinitions of the service.

ServiceRemoteFunction

class dsviper.**ServiceRemoteFunction**

Bases: object

A class used to call a remote function.

Note: Not directly instantiable.

function() → *ServiceRemoteFunctionPoolFunction*

Return the function.

pool() → *ServiceRemoteFunctionPool*

Return the pool.

service() → *ServiceRemote*

Return the service.

ServiceRemoteFunctionPool

class dsviper.**ServiceRemoteFunctionPool**

Bases: object

A class used to represent a remote function pool.

Note: Not directly instantiable.

check(*func_name: str*) → *ServiceRemoteFunctionPoolFunction*

Return a function or raise.

documentation() → str

Return the documentation.

functions() → list[*ServiceRemoteFunctionPoolFunction*]

Return the list of functions.

name() → str

Return the name.

query(*func_name: str*) → *ServiceRemoteFunctionPoolFunction* | None

Return a function or None.

uuid() → *ValueUUId*

Return the uuid.

ServiceRemoteFunctionPoolFunction

class dsviper.**ServiceRemoteFunctionPoolFunction**

Bases: object

A class used to represent a remote function.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

prototype() → *FunctionPrototype*

Return the prototype.

ServiceRemoteFunctionPoolFunctions

class dsviper.**ServiceRemoteFunctionPoolFunctions**

Bases: object

A class used to represent the functions of a pool.

Note: Not directly instantiable.

ServiceRemoteFunctionPools

class dsviper.**ServiceRemoteFunctionPools**

Bases: object

A class used to represent the remote function pools.

Note: Not directly instantiable.

ServiceRemoteAttachmentFunction

class dsviper.**ServiceRemoteAttachmentFunction**

Bases: object

A class used to call a remote attachment function.

Note: Not directly instantiable.

function() → *ServiceRemoteAttachmentFunctionPoolFunction*

Return the function.

pool() → *ServiceRemoteAttachmentFunctionPool*

Return the pool.

service() → *ServiceRemote*

Return the service.

ServiceRemoteAttachmentFunctionPool

class dsviper.**ServiceRemoteAttachmentFunctionPool**

Bases: object

A class used to represent a remote attachment function pool.

Note: Not directly instantiable.

check(*func_name: str*) → *ServiceRemoteAttachmentFunctionPoolFunction*

Return a function or raise.

documentation() → str

Return the documentation.

functions() → list[*ServiceRemoteAttachmentFunctionPoolFunction*]

Return the list of functions.

name() → str

Return the name.

query(*func_name: str*) → *ServiceRemoteAttachmentFunctionPoolFunction* | None

Return a function or None.

uuid() → *ValueUUId*

Return the uuid.

ServiceRemoteAttachmentFunctionPoolFunction

class dsviper.**ServiceRemoteAttachmentFunctionPoolFunction**

Bases: object

A class used to represent a remote attachment function.

Note: Not directly instantiable.

documentation() → str

Return the documentation.

is_mutable() → bool

Return True if the function is mutable.

prototype() → *FunctionPrototype*

Return the prototype.

ServiceRemoteAttachmentFunctionPoolFunctions

class dsviper.**ServiceRemoteAttachmentFunctionPoolFunctions**

Bases: object

A class used to represent the remote attachment functions.

Note: Not directly instantiable.

ServiceRemoteAttachmentFunctionPools

class dsviper.**ServiceRemoteAttachmentFunctionPools**

Bases: object

A class used to represent the remote attachment pools.

Note: Not directly instantiable.

Utilities

dsviper.ViperError	
dsviper.Error	A class used to represent an error raised by Viper.
dsviper.Cancelation	A class used to request a cancelation.
dsviper.Semaphore	A class used to represent a semaphore.
dsviper.SharedMemory	A class used to represent a shared memory region.
dsviper.Socket	A class used to represent a shared memory region.
dsviper.Float16	A class used to handle float16 representation.
dsviper.Fuzzer	A class used to generate a random value.
dsviper.KeyHelper	A class used to collect keys, attachments and missing attachments for a key.
dsviper.KeyNamer	A class used to get a name for a key from available attachments.

dsviper.ViperError

exception dsviper.**ViperError**

Error

class dsviper.**Error**(*module*, *domain*, *code*, *message*)

Bases: object

A class used to represent an error raised by Viper.

code() → str

Return the code.

component() → str

Return the component.

domain() → str
Return the domain.

explained() → str

-

Return the explanation of the error.

hostname() → str
Return the hostname.

message() → str
Return the message.

static parse(*description: str*) → *Error* | None
Return an Error or None.

process_name() → str
Return the process name.

static set_process_name(*name: str*) → None
Assign the name of the process reported in Error.

Cancelation

class dsviper.**Cancelation**
Bases: object

A class used to request a cancelation.

cancel() → None
request a cancelation.

requested() → bool
Return True if a cancelation is requested.

Semaphore

class dsviper.**Semaphore**
Bases: object

A class used to represent a semaphore. Use the static factory method create(…) or open(…).

Note: Not directly instantiable.

static create(*name: str*) → *Semaphore*
Create a semaphore.

static exists(*name: str*) → bool
Return True is the name exist.

name() → str
Return the name.

static open(*name: str*) → *Semaphore*
Open an existing semaphore.

post() → bool
The semaphore is unlocked.

try_wait() → bool
The semaphore is locked.

static unlink(*name: str*) → bool
Unlink the shared memory object.

wait() → bool
The semaphore is locked.

SharedMemory

class dsviper.**SharedMemory**
Bases: object

A class used to represent a shared memory region.

Note: Not directly instantiable.

address() → int
Return the address.

static create(*name: str*, *size: int*) → *SharedMemory*
Create a shared memory segment.

static exists(*name: str*) → bool
Return True is the name exist.

fd() → int
Return the file descriptor.

name() → str
Return the name.

static open(*name: str*, *size: int*) → *SharedMemory*
Open an exising shared memory segment.

size() → int
Return the size.

static unlink(*name: str*) → bool
Unlink the shared memory object.

Socket

class dsviper.**Socket**
Bases: object

A class used to represent a shared memory region.

Note: Not directly instantiable.

static create_passive_inet(*host: str*, *service: str*) → *Socket*
Create an inet passive socket.

static create_passive_local(*socket_path: str*) → *Socket*
Create a local passive socket.

Float16

class dsviper.**Float16**

Bases: object

A class used to handle float16 representation.

Note: Not directly instantiable.

static from_float(*v: float*) → int

Return the float16 as an unsigned int.

static to_float(*v: int*) → float

Return the float from an unsigned int representing a float16 storage.

Fuzzer

class dsviper.**Fuzzer**(*definitions*)

Bases: object

A class used to generate a random value.

blob_id() → *ValueBlobId* | None

Return The special uniq blob_id or None.

blob_size() → int

Return the size of the generated blob.

fuzz(*type:* Type) → *Value*

Generate and return a random value from the type.

map_size() → int

Set the size of the generated map.

set_blob_id(*blob_id:* ValueBlobId | *None* = *None*) → None

Set The special uniq blob_id if specified.

set_blob_size(*value: int*) → None

Set the size of the generated blob.

set_map_size(*value: int*) → None

Set the size of the generated map.

set_set_size(*value: int*) → None

Set the size of the generated set.

set_size() → int

Return the size of the generated set.

set_string_size(*value: int*) → None

Set the size of the generated string.

set_vector_size(*value: int*) → None

Set the size of the generated vector.

set_xarray_size(*value: int*) → None

Set the size of the generated xarray.

string_size() → int
Return the size of the generated string.

types() → list[*Type*]
Return the list of types.

vector_size() → int
Return the size of the generated vector.

xarray_size() → int
Return the size of the generated xarray.

KeyHelper

class dsviper.**KeyHelper**
Bases: object

A class used to collect keys, attachments and missing attachments for a key.

Note: Not directly instantiable.

static attachments(*type:* Type, *definitions:* DefinitionsConst) → list[*Attachment*]
Return a list of attachments.

static collect_keys(*type_key:* TypeKey, *attachment_getting:* AttachmentGetting) → *ValueSet*
Return a set of key.

static missing_attachments(*key:* ValueKey, *attachment_getting:* AttachmentGetting) → list[*Attachment*]
Return the list of missing attachments.

KeyNamer

class dsviper.**KeyNamer**(*definitions*)
Bases: object

A class used to get a name for a key from available attachments.

name(*key:* ValueKey, *attachment_getting:* AttachmentGetting) → str | None
Return the name or None.

smart_name(*key:* ValueKey, *attachment_getting:* AttachmentGetting) → str | None
Return the name or None.

3.3.11 Overview

Domain	Description
Type System	Type system (`Type`, `TypeVector`, `TypeMap`, etc.)
Values	Value instances (`Value`, `ValueString`, `ValueVector`, etc.)
Attachments	Attachments (`Attachment`, `AttachmentGetting`, `AttachmentMutating`, etc.)
Database	Persistence (`Database`, `DatabaseSQLite`, etc.)
Commit System	Versioned persistence (`CommitDatabase`, `CommitState`, `AttachmentMutating`, etc.)
Binary Data (Blobs)	Binary data (`BlobArray`, `BlobPack`, `BlobLayout`, etc.)
Serialization	Encoding/decoding (`Codec`, `StreamEncoder`, etc.)
DSM Introspection	Model introspection (`DSMBuilder`, `DSMDefinitions`, etc.)
HTML Rendering	HTML rendering (`DocumentNode`, `Html`, etc.)
Core Utilities	Core utilities (`Definitions`, `NameSpace`, `Path`, etc.)

3.3.12 Reference Applications

To see the full dsviper API in action, explore `cdbe.py` (CDB Editor) and `dbe.py` (Database Editor) in the `tools/` directory. These Qt/PySide6 applications demonstrate the complete capabilities of dsviper:

Capability	API Used
Type System	`Type`, `TypeVector`, `TypeMap` for schema definition and validation
Value Handling	`Value`, `ValueStructure`, `ValueKey` for typed data manipulation
Versioned Persistence	`CommitDatabase`, `CommitState` for DAG-based history
Mutation Context	`CommitMutableState`, `AttachmentMutating` for isolated evaluation
Path-Based Mutations	`Path`, `AttachmentMutating.update()` for multiplayer editing
Undo/Redo	`CommitStore` for application-level undo with full state restoration
Synchronization	`CommitSynchronizer` for fetch/push/sync with remote servers
Live Sync	`CommitStoreNotifying` for real-time multi-user updates
Binary Assets	`BlobGetting`, `ValueBlobId` for large binary data management
Schema Introspection	`DSMDefinitions`, `Definitions` for runtime type discovery
HTML Rendering	`Html.dsm_definitions()`, `Html.value()` for visual inspection
Error Handling	`ViperError`, `Error.parse()` for structured error reporting
Logging	`Logging`, `LoggerConsole` for operation tracing

These tools serve as comprehensive examples for building your own dsviper applications.

3.4 Toolchain Manual

This manual covers the complete development toolkit for DSM and Viper:

- **dsm_util.py** - Model validation and database creation
- **Kibo** - Code generation from DSM definitions
- **IDE Integration** - VS Code and JetBrains plugins
- **Database Editors** - GUI tools (cdbe.py, dbe.py)
- **Server & Admin** - Network deployment and administration

3.4.1 dsm_util.py

`dsm_util.py` is the command-line tool for working with DSM definitions. It validates models, creates databases, and generates Python packages.

Commands

Command	Description
`check`	Validate DSM syntax
`create_commit_database`	Create a versioned database
`create_database`	Create a simple database
`create_python_package`	Generate Python package
`encode`	Convert DSM to binary format
`decode`	Convert binary to DSM format

Check Syntax

Validate DSM definitions and report errors:

```
# Check a single file
python3 tools/dsm_util.py check model.dsm

# Check a folder of DSM files
python3 tools/dsm_util.py check definitions/
```

Error Format

Errors are reported in the standard format `<file>:<line>:<column>:<message>`:

```
model.dsm:10:5: The type 'strin' is unknown.
model.dsm:14:12: The type K='user' in attachment<K, D> 'identity' must be a concept.
```

This format integrates with most IDEs for click-to-navigate.

Create Commit Database

Create an empty versioned database with embedded definitions:

```
python3 tools/dsm_util.py create_commit_database model.dsm model.cdb
```

A CommitDatabase:

- Stores all mutations as commits
- Maintains full history (DAG)
- Embeds definitions for self-contained distribution

Create Database

Create a simple (non-versioned) database:

```
python3 tools/dsm_util.py create_database model.dsm model.db
```

Use this for simpler applications that don't need history.

Create Python Package

Generate a complete Python package from DSM definitions:

```
python3 tools/dsm_util.py create_python_package model.dsm --repos /path/to/repos
```

Generated Package Contents

Module	Description
`model.definitions`	Type definitions for Viper
`model.data`	Classes for concepts, structures, enums
`model.attachments`	Attachment accessors with type hints
`model.database`	Database API with type hints
`model.value_types`	Type mappings

Usage

```
>>> import model.attachments as ma
>>> import model.data as md

# Use generated classes
>>> key = md.Tuto_UserKey.create()
>>> login = md.Tuto_Login()
>>> login.nickname = "alice"

# Use generated accessors
>>> ma.tuto_user_login_set(attachment_mutating, key, login)
```

Options

Option	Description
`--repos`	Path to Digital Substrate repositories
`--wheel`	Generate pyproject.toml for wheel building
`--kibo`	Path to kibo JAR file
`--template`	Path to template folder

Encode to Binary

Convert DSM definitions to binary format (`.dsmb`):

```
python3 tools/dsm_util.py encode model.dsm model.dsmb
```

The binary format is:

- Faster to load
- Used by Kibo code generator
- Platform-independent

Decode from Binary

Convert binary definitions back to DSM format:

```
python3 tools/dsm_util.py decode model.dsmb decoded_model.dsm
```

Workflow Example

```
# 1. Write your DSM model
vim model.dsm

# 2. Check syntax
python3 tools/dsm_util.py check model.dsm

# 3. Create database
python3 tools/dsm_util.py create_commit_database model.dsm model.cdb
```

(continues on next page)

(continued from previous page)

```
# 4. Generate Python package
python3 tools/dsm_util.py create_python_package model.dsm

# 5. Use in Python
python3 -c "import model; print(model)"
```

What's Next

- *Kibo* - The code generator
- *IDE Integration* - VS Code and JetBrains

3.4.2 Kibo

Kibo is the code generator that transforms DSM definitions into C++ and Python infrastructure code.

Overview

```
DSM Definitions  →  Kibo  →  Generated Code
   (273 lines)      ↓       (15,592 lines)
                Templates
```

Kibo uses StringTemplate files (`.stg`) to generate code from DSM models.

Synopsis

```
java -jar kibo-1.2.7.jar \
  -c [converter] \
  -n [namespace] \
  -d [definitions.dsmb] \
  -t [template] \
  -o [output]
```

Options

Option	Description
`-c, --converter`	Target language: `cpp` or `python`
`-n, --namespace`	Namespace for generated files
`-d, --definitions`	Path to binary definitions (`.dsmb`)
`-t, --template`	Template directory or file
`-o, --output`	Output directory or file
`-q, --quiet`	Disable output messages
`-h, --help`	Show help
`-v, --version`	Show version

C++ Templates

Available template groups in `templates/cpp/`:

Template	Generated Files	Purpose
Data	*_Data.hpp/cpp	Types for enum, struct, concept, club
Model	*_Definitions.hpp/cpp	DSM definitions, paths, fields
Attachments	*_Attachments.hpp/cpp	Attachment accessors
Database	*_Database*.hpp/cpp	SQLite persistence layer
Stream	*_Reader/Writer.hpp/cpp	Binary serialization
Json	*_JsonEncoder/Decoder.hpp/cpp	JSON serialization
ValueType	*_ValueType.hpp/cpp	Viper type mappings
ValueCodec	*_ValueEncoder/Decoder.hpp/cpp	Bridge C++ classes (static) ↔ Value (dynamic)
ValueHasher	*_ValueHasher.hpp/cpp	Content hashing for values
FunctionPool	*_FunctionPools.hpp/cpp	Pure function bindings
FunctionPoolRemote	*_FunctionPoolsRemote.hpp/cpp	Remote function call API
AttachmentFunctionPool	*_AttachmentFunctionPools.hpp/cpp	Stateful pool bindings
AttachmentFunctionPool-Remote	*_AttachmentFunctionPoolsRemote.hpp/cpp	Remote stateful calls
AttachmentFunction-Pool_Attachments	*_AttachmentFunctionPools_Attachm hpp/cpp	Pool exposing attachment API
Python	*_Python.hpp/cpp	Python module constants for types and paths
Fuzz	*_Fuzz.hpp/cpp	Random value generation
Test	*_Test*.hpp/cpp	Unit tests
TestApp	*_TestApp.cpp	Test application entry points

Python Templates

Available template groups in `templates/python/`:

Template	Generated File	Purpose
package/__init__.py	__init__.py	Package initialization
package/definitions	definitions.py	Viper type definitions
package/data	data.py	Concept, structure, enum classes
package/attachments	attachments.py	Attachment API wrappers
package/database_attachments	database_attachments.py	Database attachment API wrappers
package/path	path.py	Field path constants
package/value_type	value_type.py	Type mappings
package/function_pools	function_pools.py	Function pool wrappers
package/attachment_function_pools	attachment_function_pools.py	Stateful pool wrappers
package/function_pool_remotes	function_pool_remotes.py	Remote function call wrappers
package/attachment_function_pool_remotes	attachment_function_pool_remo py	Remote stateful call wrappers
wheel/pyproject.toml	pyproject.toml	Wheel configuration

Usage Examples

Generate All C++ Features

```
# From generate.py
templates = [
    'Model', 'Data',
    'Attachments', 'Stream', 'Json',
    'Database',
    'ValueType', 'ValueCodec', 'ValueHasher',
    'Fuzz', 'Test'
]

for template in templates:
    subprocess.run([
        'java', '-jar', 'kibo-1.2.7.jar',
        '-c', 'cpp',
        '-n', 'MyApp',
        '-d', 'model.dsmb',
        '-t', f'templates/cpp/{template}',
        '-o', 'src/MyApp'
    ])
```

Generate Python Package

```
subprocess.run([
    'java', '-jar', 'kibo-1.2.7.jar',
    '-c', 'python',
    '-n', 'mymodel',
    '-d', 'model.dsmb',
    '-t', 'templates/python/package',
    '-o', 'python/mymodel'
])
```

Complete generate.py Example

A production-ready script combining all generation steps:

```
#!/usr/bin/env python3
"""Complete code generation script for a Viper project."""
import subprocess
import argparse
import base64
import zlib
from dsviper import DSMBuilder

# Configuration
JAR = 'tools/kibo-1.2.7.jar'
KIBO = ['java', '-jar', JAR]
TEMPLATES_CPP = 'templates/cpp'
TEMPLATES_PY = 'templates/python/package'
```

(continues on next page)

(continued from previous page)

```
def check_report(report):
    """Exit on parse errors."""
    if report.has_error():
        for err in report.errors():
            print(repr(err))
        exit(1)

def generate_cpp(namespace, dsmb, template, output):
    """Generate C++ code from a template."""
    subprocess.run(KIBO + [
        '-c', 'cpp', '-n', namespace,
        '-d', dsmb, '-t', f'{TEMPLATES_CPP}/{template}',
        '-o', output
    ])

def generate_resource(definitions, output):
    """Generate embedded C++ resource."""
    blob = definitions.encode()
    with open(output, 'w') as f:
        f.write(blob.embed("definitions"))

def generate_python(name, dsmb, definitions, output):
    """Generate Python package with embedded definitions."""
    subprocess.run(KIBO + [
        '-c', 'python', '-n', name,
        '-d', dsmb, '-t', TEMPLATES_PY,
        '-o', output
    ])
    # Embed definitions as base64
    blob = definitions.encode()
    encoded = base64.b64encode(zlib.compress(blob))
    with open(f'{output}/resources.py', 'w') as f:
        f.write(f"B64_DEFINITIONS = {encoded}")

# Parse DSM
builder = DSMBuilder.assemble("definitions/MyApp")
report, dsm_defs, definitions = builder.parse()
check_report(report)

# Save binary definitions
with open("MyApp.dsmb", "wb") as f:
    f.write(dsm_defs.encode())

# Generate C++ (standard templates)
for template in ['Model', 'Data', 'Attachments', 'Stream', 'Database',
                 'ValueType', 'ValueCodec', 'ValueHasher', 'FunctionPool']:
    generate_cpp('MyApp', 'MyApp.dsmb', template, 'src/MyApp')

# Generate embedded resource
generate_resource(definitions, 'src/MyApp/MyApp_Resources.hpp')

# Generate Python package
```

(continues on next page)

(continued from previous page)

```
generate_python('myapp', 'MyApp.dsmb', definitions, 'python/myapp')

print("Generation complete!")
```

Workflow

Step 1: Parse and Encode DSM

```
from dsviper import DSMBuilder

# Parse DSM file
builder = DSMBuilder.assemble("model.dsm")
report, dsm_defs, definitions = builder.parse()

# Check for errors
if report.has_errors():
    print(report.errors())
    exit(1)

# Save binary format
with open("model.dsmb", "wb") as f:
    f.write(dsm_defs.encode().encoded())
```

Step 2: Generate Embedded Definitions

```
# Generate C++ resource for Definitions.cpp
blob = definitions.encode()
with open("Resources.hpp", "w") as f:
    f.write(blob.embed("definitions"))
```

Step 3: Run Kibo

```
java -jar kibo-1.2.7.jar -c cpp -n MyApp -d model.dsmb -t templates/cpp/Data -o src/
```

Step 4: Compile and Link

The generated code requires linking against the Viper runtime.

Amplification Metrics

Project	DSM Lines	Generated Lines	Ratio
Graph Editor	273	15,592	57x
Raptor Editor	2,003	126,182	63x

Kibo typically generates 57-63x more code than the input DSM.

What's Next

- *IDE Integration* - VS Code and JetBrains
- *Templates* - Understanding templated features

3.4.3 IDE Integration

Two IDE extensions support DSM development with syntax highlighting, validation, and code completion.

IDE	Extension	Strengths
VS Code	`ds-dsm`	Lightweight, snippets, problem matcher
JetBrains	DSM Language	Full IDE: completion, navigation, refactoring

VS Code Extension

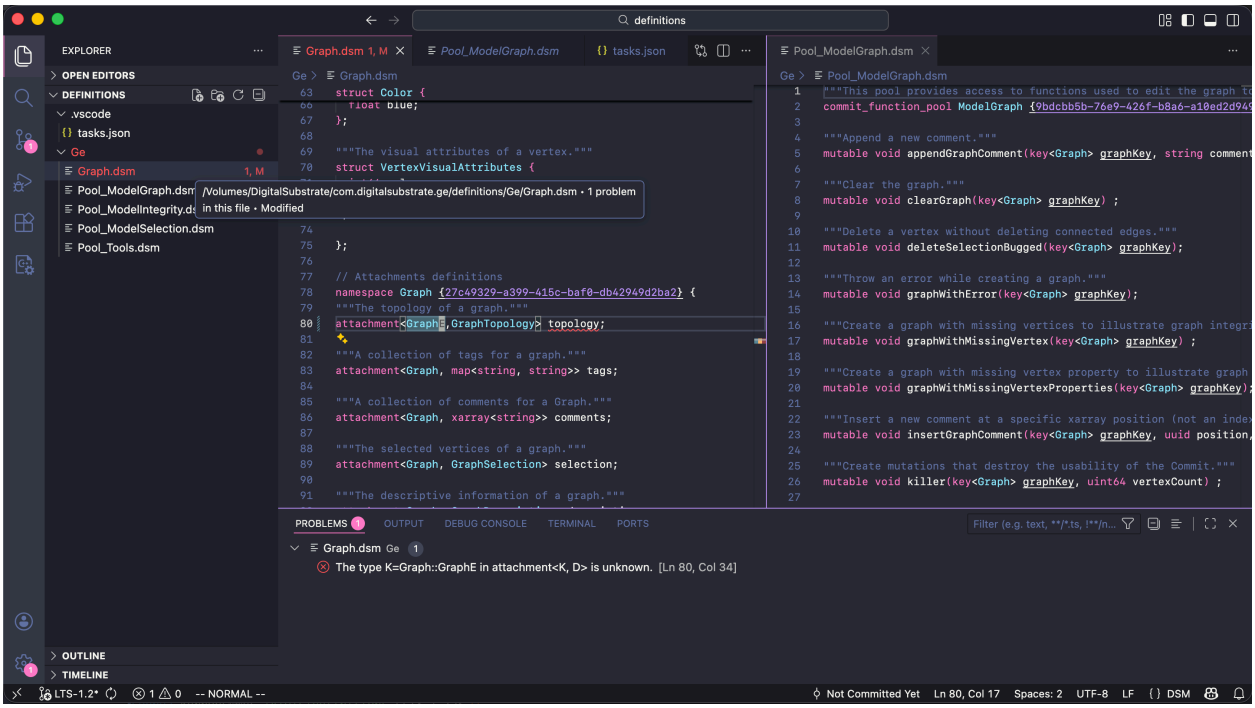


Fig. 1: VS Code with DSM syntax highlighting and problem matcher.

Installation

The VS Code extension is published on the VS Code Marketplace.

1. Open Visual Studio Code
2. Open the Extensions view: `Cmd+Shift+X` (macOS) or `Ctrl+Shift+X` (Windows/Linux)
3. Search for **DSM Syntax Highlighter** (publisher: *DigitalSubstrate*)
4. Click **Install**

Alternatively, install from the command line:

```
code --install-extension DigitalSubstrate.ds-dsm
```

Source repository: github.com/digital-substrate/dsm-vscode.

Syntax Highlighting

The extension highlights:

- Keywords (`namespace`, `concept`, `struct`, `attachment`)
- Types (`string`, `int64`, `vector<T>`)
- UUIDs, comments, docstrings

Snippets

Type a prefix and press Tab to expand:

Structures:

Prefix	Expansion
`namespace`	Namespace with UUID placeholder
`concept`	`concept Name;`
`struct`	Structure template
`enum`	Enumeration template
`attachment`	`attachment<Concept, Struct> name;`
`function_pool`	Function pool with UUID
`attachment_function_pool`	Attachment function pool

Types:

Prefix	Expansion
`key`	`key<T>`
`optional`	`optional<T>`
`vec`	`vec<T, size>`
`map`	`map<K, V>`
`set`	`set<T>`

Advanced:

Prefix	Expansion
`isa`	`concept Derived is a Base;`
`opt-key`	`optional<key<Concept>> fieldKey;`
`mutable`	Mutable function with docstring

Build Task Configuration

Create `.vscode/tasks.json` to enable syntax checking on build:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Check DSM Syntax",
      "type": "shell",
      "command": "python",
      "args": [
        "../tools/dsm_util.py",
        "check",
        "${file}"
      ],
      "presentation": {
        "reveal": "never",
        "revealProblems": "onProblem"
      },
      "problemMatcher": "$dsm",
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Using the Build Task

- Press `Cmd+Shift+B` (macOS) or `Ctrl+Shift+B` (Windows/Linux)
- Errors appear in the Problems panel with click-to-navigate

Optional: Theme Customization

Dracula Theme (Recommended)

1. Open Command Palette
2. Run: `Browse Color Theme in MarketPlace`
3. Select: `Dracula`

Token Customization

Add to your `settings.json`:

```
{
  "editor.tokenColorCustomizations": {
    "textMateRules": [
      {
        "scope": "variable.key.dsm",
        "settings": {
```

(continues on next page)

(continued from previous page)

```
          "fontStyle": "underline"
        }
      },
      {
        "scope": "constant.uuid.dsm",
        "settings": {
          "fontStyle": "underline"
        }
      },
      {
        "scope": "constant.enum.dsm",
        "settings": {
          "fontStyle": "italic"
        }
      }
    ]
  }
}
```

This customization:

- Underlines key types
- Underlines UUIDs
- Italicizes enum cases

JetBrains Plugin

For IntelliJ IDEA, CLion, PyCharm, and other JetBrains IDEs.

Installation

The JetBrains plugin is published on the JetBrains Marketplace.

1. Open your JetBrains IDE (IntelliJ IDEA, PyCharm, CLion, etc.)
2. Go to: `Settings/Preferences > Plugins > Marketplace`
3. Search for **DSM** (publisher: *Digital Substrate*)
4. Click **Install**
5. Restart the IDE when prompted

Source repository: github.com/digital-substrate/dsm-jetbrains.

Key Features for DSM Development

Context-Aware Completion:

The plugin suggests different types based on cursor position:

- Inside `attachment<▸>` → suggests concepts and clubs only
- Inside `attachment<Graph, ▸>` → suggests all types (structs, primitives)
- Inside `key<▸>` → suggests concepts only (not structs)

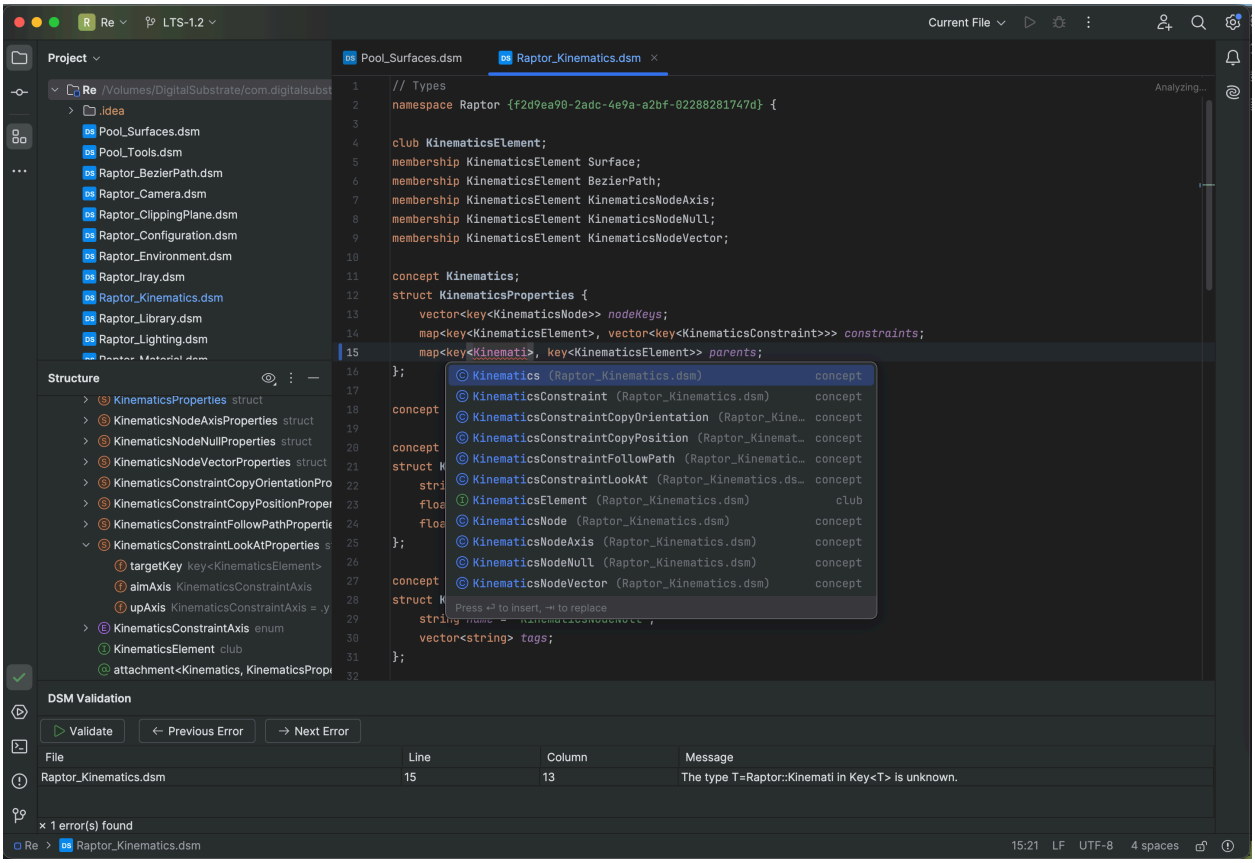


Fig. 2: JetBrains plugin showing syntax highlighting, code completion, Structure view, and DSM Validation panel.

Navigation:

Action	macOS	Windows/Linux
Go to Definition	`Cmd+Click`	`Ctrl+Click`
Find Usages	`Alt+F7`	`Alt+F7`
Go to Symbol	`Cmd+Alt+O`	`Ctrl+Alt+N`

Validation:

Action	Shortcut
Validate DSM	`Ctrl+Alt+V`
Next Error	`Alt+F2`
Previous Error	`Alt+Shift+F2`

Refactoring:

- **Rename** (`Shift+F6`): Renames across all files
- **Reformat** (`Cmd+Alt+L` / `Ctrl+Alt+L`): Consistent formatting

Development Workflow

The typical DSM development cycle:

```
Write DSM  →  Validate  →  Fix Errors  →  Generate Code
   |            |             |              |
   ▼            ▼             ▼              ▼
 IDE        Cmd+Shift+B    Click error    generate.py
 Snippets   or Ctrl+Alt+V  to navigate    or dsm_util.py
```

Project Structure

```
my_project/
├── definitions/
│   ├── Model.dsm
│   └── Pools.dsm
├── src/generated/      # Kibo output (C++)
├── python/mymodel/     # Kibo output (Python)
└── generate.py         # Code generation script
```

What's Next

- *Wheels* - Creating Python wheels
- *Templates* - Understanding templated features

3.4.4 Database Editors

Viper provides two GUI applications for exploring and managing databases:

Tool	Purpose	Use When
cdbe.py	CommitDatabase Editor	Working with versioned data (undo/redo, sync, history)
dbe.py	Database Editor	Simple exploration without commit history

Both tools are built with PySide6 and share common components from `ds_components/`.

---

cdbe.py - CommitDatabase Editor

Full-featured GUI for exploring and managing CommitDatabases with version history.

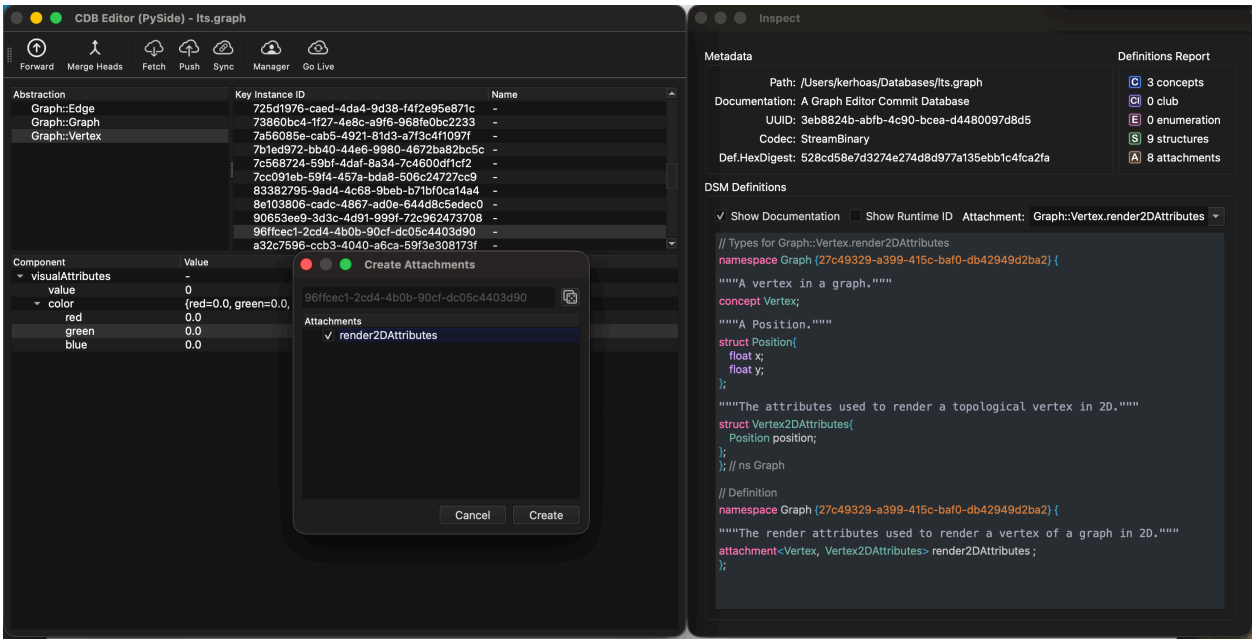


Fig. 3: cdbe.py showing commit history, document browser, and database inspector.

Launch

```
python3 tools/cdbe.py                    # Open file dialog
python3 tools/cdbe.py project.cdb        # Open specific database
```

Panels

Access panels via keyboard shortcuts:

Panel	Shortcut	Description
Commits	`Cmd+1`	Browse commit history (DAG visualization)
Documents	`Cmd+2`	Browse documents by attachment
Program	`Cmd+3`	View Program in selected commit
Settings	`Cmd+4`	Configure synchronization source
Undo	`Cmd+5`	Undo/redo stack visualization
Sync Log	`Cmd+6`	Synchronization activity log
Blobs	`Cmd+7`	Browse blob storage
Actions	`Cmd+8`	Pending actions queue

Synchronization

cdbe.py supports synchronizing with a remote CommitDatabase server.

Connection Settings:

- TCP: `host:port` (e.g., `server.local:54321`)
- Unix socket: `/path/to/socket`

Sync Operations:

Action	Description
Fetch	Pull commits from remote server
Push	Push local commits to remote
Sync	Full sync (Fetch + Push)

Live Mode:

Enable automatic synchronization at configurable intervals:

- **Live Mode**: Auto-sync on timer
- **Manager Mode**: Auto-merge heads during live sync

Inspecting Commits

1. Open the **Commits** panel (`Cmd+1`)
2. Click a commit to select it
3. Open **Commands** panel (`Cmd+3`) to see mutations
4. Double-click a document to inspect its content

Undo/Redo

cdbe.py maintains an undo stack for local operations:

- **Undo**: `Cmd+Z`
- **Redo**: `Cmd+Shift+Z`

View the undo stack in the **Undo** panel (`Cmd+5`).

dbe.py - Database Editor

Simplified GUI for exploring Database files (without commit history).

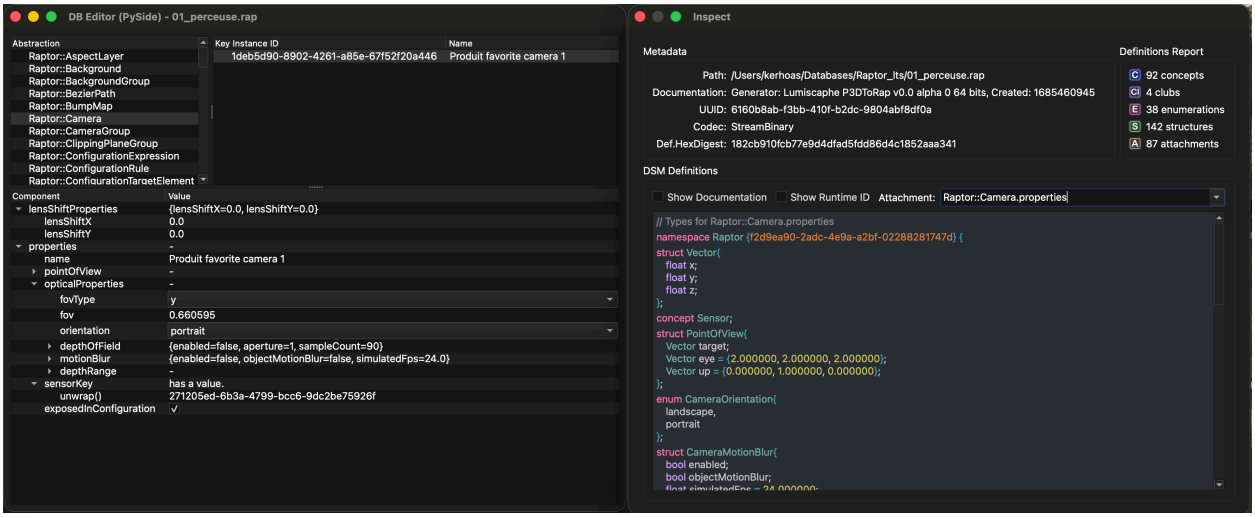


Fig. 4: dbe.py showing document browser and database inspector.

Launch

```
python3 tools/dbe.py                    # Open file dialog
python3 tools/dbe.py project.vpr        # Open specific database
```

Features

Feature	Description
Document Browser	Browse documents by attachment
Blob Viewer	Inspect binary blob content
Database Inspector	View UUID, codec, definitions
Network Connection	Connect to remote Database server

When to Use dbe.py vs cdbe.py

Scenario	Tool
Explore versioned data with history	cdbe.py
Simple database inspection	dbe.py
Synchronize with remote server	cdbe.py
Debug blob storage	Either

ds_components - Shared GUI Library

Both editors use `ds_components/`, a PySide6 widget library:

Component	Description
`DSDocumentsCommitStore`	Document browser for CommitStore
`DSDocumentsDatabasing`	Document browser for Database
`DSCommitsDialog`	Commit history panel
`DSValueProgramDialog`	Program viewer
`DSInspectDialog`	Database inspector
`DSBlobsDialog`	Blob storage viewer
`DSSettings`	User preferences management

These components can be reused in custom applications built on Viper.

What's Next

- *Server* - Running a CommitDatabase server
- *dsm_util* - Creating databases from DSM

3.4.5 Database Server & Administration

Tools for running CommitDatabase as a network service and performing administrative operations.

Tool	Type	Purpose
commit_database_server.py	Server	Expose CommitDatabase over RPC
commit_admin.py	CLI	Database administration (reset, sync, merge)

---

commit_database_server.py

RPC server that exposes a CommitDatabase over the network, allowing multiple clients to synchronize with a central database.

Basic Usage

```
# Start server with default settings (TCP port 54321)
python3 tools/commit_database_server.py project.cdb

# Custom host and port
python3 tools/commit_database_server.py --host 0.0.0.0 --port 54322 project.cdb

# Unix socket (for local IPC, better performance)
python3 tools/commit_database_server.py --socket-path /tmp/project.sock project.cdb

# Verbose logging
python3 tools/commit_database_server.py -vvv project.cdb
```

Options

Option	Description	Default
`--host`	Bind address	`0.0.0.0`
`--port`	TCP port	`54321`
`--socket-path`	Unix socket path (alternative to TCP)	-
`-v`	Verbosity: `-v` critical, `-vv` info, `-vvv` debug	quiet

Graceful Shutdown

Press `Ctrl+C` to stop the server gracefully. Active connections will be closed and the database will be properly finalized.

Client Connection

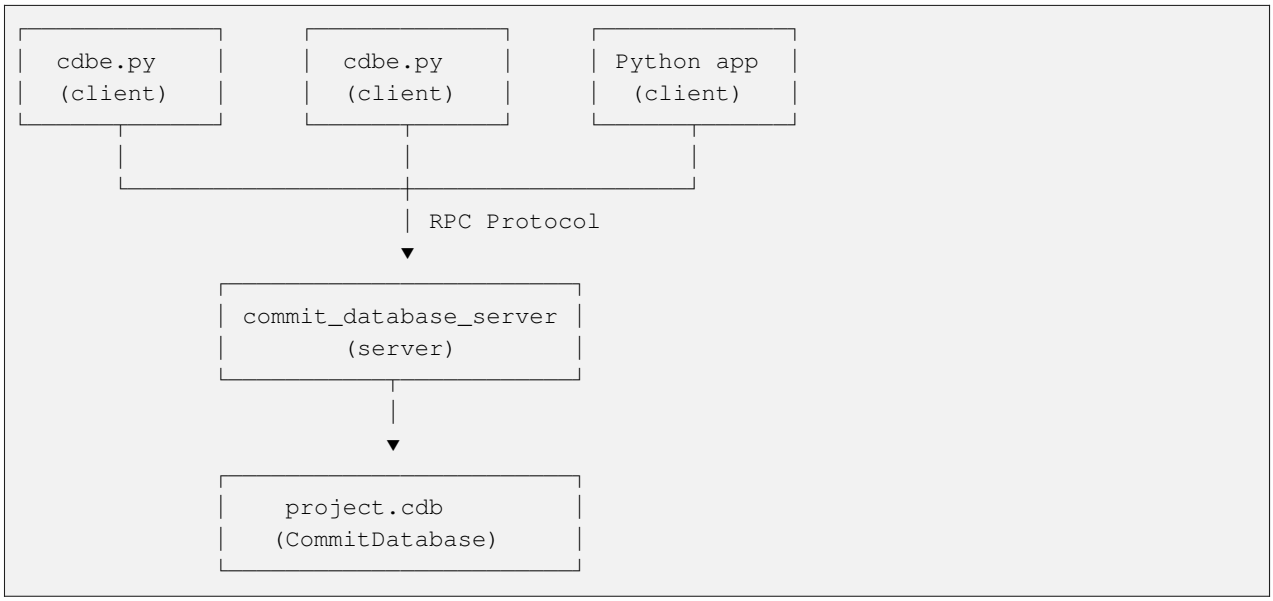
Clients connect using `CommitDatabase.connect()` or `CommitDatabase.connect_local()`:

```
from dsviper import CommitDatabase

# TCP connection
db = CommitDatabase.connect("server.local", "54321")

# Unix socket
db = CommitDatabase.connect_local("/tmp/project.sock")
```

Architecture



commit_admin.py

Administration tool for CommitDatabase operations: reset, sync, and merge heads.

Reset Database

Reset a database to its initial commit, deleting all history:

```
python3 tools/commit_admin.py --database project.cdb reset
```

Warning

This operation is destructive and cannot be undone. All commits except the initial commit will be deleted.

Reduce Heads (Merge)

When multiple clients commit concurrently, the database may have multiple "heads" (concurrent streams). Use `reduce_heads` to merge them:

```
# Single merge pass
python3 tools/commit_admin.py --database project.cdb reduce_heads

# Continuous merge (loop mode)
python3 tools/commit_admin.py --database project.cdb reduce_heads \
    --loop --update-interval 5
```

Sync Local with Remote

Synchronize a local database with a remote server:

```
# Single sync pass
python3 tools/commit_admin.py --host server.local --port 54321 sync local.cdb

# Continuous sync (loop mode)
python3 tools/commit_admin.py --host server.local sync local.cdb \
    --loop --update-interval 2

# Via Unix socket
python3 tools/commit_admin.py --socket-path /tmp/project.sock sync local.cdb
```

Options Reference

Option	Description
`--database`	Local database path
`--host`	Remote server hostname
`--port`	Remote server port (default: 54321)
`--socket-path`	Unix socket path
`--loop`	Run continuously
`--update-interval`	Seconds between iterations (default: 1)
`--blob-data-size`	Max blob pack size in MB (default: 25)

Sub-commands

Command	Description
`reset`	Reset to first commit (deletes all history)
`reduce_heads`	Merge multiple heads into one
`sync`	Synchronize local database with source

What's Next

- *Editors* - GUI tools for database exploration
- *dsm_util* - Creating databases from DSM

3.4.6 Python Wheels

This chapter covers creating distributable Python wheels from your DSM model.

What is a Wheel?

A wheel is Python's standard distribution format. It packages your generated code for easy installation via pip.

Quick Start

Step 1: Generate with Wheel Option

```
python3 tools/dsm_util.py create_python_package --wheel model.dsm
```

This creates:

- Generated Python package in `model/`
- Template `pyproject.toml`

Step 2: Edit pyproject.toml

Customize the generated `pyproject.toml`:

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"

[project]
name = "model"
version = "1.0.0"
description = "Python wrapper for model"
authors = [
    {name = "Your Name", email = "you@example.com"}
]
maintainers = [
    {name = "Your Name", email = "you@example.com"}
]
readme = "README.rst"
```

(continues on next page)

(continued from previous page)

```
requires-python = ">=3.14"
license = {text = "Proprietary"}
classifiers = [
    "Development Status :: 5 - Production/Stable",
    "Programming Language :: Python :: 3",
    "Intended Audience :: Developers",
    "Topic :: Software Development",
]
keywords = ["model"]

[tool.setuptools.packages.find]
include = ["model"]

[tool.setuptools.package-data]
model = ["*.py", "resources/*.bin"]
```

Step 3: Build the Wheel

```
pip3 wheel .
```

This creates: `model-1.0.0-py3-none-any.whl`

Step 4: Install

```
pip3 install --force-reinstall model-1.0.0-py3-none-any.whl
```

Step 5: Use

```
>>> import model
>>> from model.data import Tuto_UserKey, Tuto_Login
>>> key = Tuto_UserKey.create()
```

Package Structure

After generation, your package looks like:

```
model/
├── __init__.py
├── data.py              # Concept, struct, enum classes
├── commit.py            # Commit API
├── database.py          # Database API
├── definitions.py       # Type definitions
├── value_types.py       # Type mappings
├── path.py              # Field paths
└── resources/
    └── definitions.bin  # Embedded definitions
```

Dependencies

Your wheel depends on `dsviper`. Add to `pyproject.toml`:

```
[project]
dependencies = [
    "dsviper>=1.2.0"
]
```

Distribution

Local Installation

```
pip3 install model-1.0.0-py3-none-any.whl
```

Private PyPI

Upload to a private PyPI server:

```
pip3 install twine
twine upload --repository-url https://your-pypi.example.com model-1.0.0-py3-none-any.
↪whl
```

Requirements File

Add to `requirements.txt`:

```
dsviper>=1.2.0
./wheels/model-1.0.0-py3-none-any.whl
```

Version Management

Update version in `pyproject.toml` for each release:

```
[project]
version = "1.1.0"  # Bump for new releases
```

Follow semantic versioning:

- **Major** (2.0.0): Breaking changes to DSM model
- **Minor** (1.1.0): New concepts, structures, or attachments
- **Patch** (1.0.1): Bug fixes, documentation

What's Next

- *Templates* - Understanding templated features

3.4.7 Templated Features

Templated features are code templates that Kibo uses to generate infrastructure code from DSM definitions.

Understanding Templates

A templated feature is like a giant code snippet where the DSM definitions are injected and recursively consumed to generate repetitive code that implements a feature for a data model. Kibo knows how to map DSM primitives (bool, int8, float…) and generic collections (vector<T>, map<K, V>) to standard C++ types and to the Viper type and value system.

```
DSM Model + Template → Generated Code
```

Templated features may depend on other templated features and form an **ecosystem**. For example, the `Database` feature depends on `Attachments` and `Model`.

C++ Template Categories

Templates in `templates/cpp/` are organized by feature:

Core Types

Template	Purpose
`Data`	Type implementations for enum, struct, concept, club
`Model`	DSM definitions, paths, fields

Persistence

Template	Purpose
`Attachments`	Attachment accessors
`Database`	SQLite persistence layer

Serialization

Template	Purpose
`Stream`	Binary encoder/decoder
`Json`	JSON encoder/decoder
`ValueCodec`	Bridge between static C++ and dynamic Viper values
`ValueHasher`	Content hashing for values
`ValueType`	Viper type mappings

Function Pools

Template	Purpose
`FunctionPool`	Pure function bindings
`FunctionPoolRemote`	Remote function call API
`AttachmentFunctionPool`	Stateful function bindings
`AttachmentFunctionPoolRemote`	Remote stateful calls
`AttachmentFunctionPool_Attachments`	Pool exposing attachment API

Python Integration

Template	Purpose
`Python`	Python module constants for types and paths

Testing

Template	Purpose
`Fuzz`	Random value generation
`Test`	Unit tests for fuzzing, JSON, streaming, storing
`TestApp`	Test application entry points

Python Template Categories

Templates in `templates/python/package/`:

Template	Generated File	Purpose
`__init__.py`	`__init__.py`	Package initialization
`definitions`	`definitions.py`	Type definitions for Viper
`data`	`data.py`	Classes for concepts, clubs, enums, structures
`attachments`	`attachments.py`	Attachment API wrappers
`database_attachments`	`database_attachments.py`	Database attachment API wrappers
`path`	`path.py`	Field path constants
`value_type`	`value_type.py`	Dynamic type definitions
`function_pools`	`function_pools.py`	Function pool wrappers
`attachment_function_pools`	`attachment_function_pools.py`	Stateful pool wrappers
`function_pool_remotes`	`function_pool_remotes.py`	Remote function call wrappers
`attachment_function_pool_remotes`	`attachment_function_pool_remo py`	Remote stateful call wrappers
`wheel/pyproject.toml`	`pyproject.toml`	Minimal pyproject.toml to build a wheel

All Python features are grouped into a single Python package.

Generation Strategy

When using templated features, you must decide **what to generate** and **where to put** the generated code. Each project uses generated code for different needs:

- Low-level serialization to insert in a framework.
- Unit tests to insert in the test infrastructure.
- A part of a feature to insert in a framework.
- A part of a feature to insert in an application.
- A Python module embedded in a C++ application.
- A Python package embedded in a C++ application.
- A Python wheel to distribute.

Each project has a `generate.py` script that describes these choices.

Note

You can also create your own generation tool using the C++ Viper API, or use custom rules in your build system (CMake, etc.).

Example: exp project

The exp project generates a framework and a Python package:

```
# Extracted from exp/generate.py
if arguments.cpp:
    # Generate a framework
    render_templates(namespace=NAMESPACE, dsmb_path=DSMB_PATH, output=PROJECT)
    generate_resource(definitions=DEFINITIONS,
                      output=f'{NAMESPACE}/{NAMESPACE}_Resources.hpp')

    # Generate application for testing the generated code.
    generate(namespace=NAMESPACE, dsmb_path=DSMB_PATH, template='TestApp', output=".")

if arguments.package:
    # Generate the Python package
    generate_package(name='exp', dsmb_path=DSMB_PATH, definitions=DEFINITIONS,
                     output=f'python/exp')
```

Example: Raptor Editor project

A larger project distributes generated code across multiple targets:

```
# Extracted from com.digitalsubstrate.red/generate.py
if arguments.red:
    # Generate the framework Raptor
    render_templates(namespace=NAMESPACE, dsmb_path=DSMB_PATH, output=f'src/
↪{NAMESPACE}')
    generate_resource(definitions=DEFINITIONS,
                      output=f'src/{NAMESPACE}/{NAMESPACE}_Resources.hpp')

if arguments.logic:
    # Generate a part of the RaptorLogic framework
    generate_logic(namespace='RaptorLogic', dsmb_path=DSMB_PATH, template='RaptorLogic
↪',
                   output=f'src/RaptorLogic')

if arguments.editor:
    # Generate a part of the application
    generate(namespace='RE', dsmb_path=DSMB_PATH, template='Python',
             output=f'RaptorEditor/RaptorEditor')

if arguments.python:
    # Generate the Python package embedded in the application
    generate_package(name='red', dsmb_path=DSMB_PATH, definitions=DEFINITIONS,
```

(continues on next page)

(continued from previous page)

```
                    output=f'RaptorEditor/RaptorEditor/Scripts/red')
```

Generation Workflow

1. Prepare Binary Definitions

```
from dsviper import DSMBuilder

builder = DSMBuilder.assemble("model.dsm")
report, dsm_defs, definitions = builder.parse()

# Save binary format for Kibo
with open("model.dsmb", "wb") as f:
    f.write(dsm_defs.encode().encoded())
```

2. Generate Embedded Resource

```
# For C++ Model/Definitions.cpp
blob = definitions.encode()
with open("Resources.hpp", "w") as f:
    f.write(blob.embed("definitions"))
```

3. Call Kibo

```
import subprocess


def generate(namespace, dsmb_path, template, output):
    subprocess.run([
        'java', '-jar', 'tools/kibo-1.2.7.jar',
        '-c', 'cpp',
        '-n', namespace,
        '-d', dsmb_path,
        '-t', f'templates/cpp/{template}',
        '-o', output
    ])


# Generate all features
templates = ['Model', 'Data', 'Attachments', 'Stream', 'Json', 'Database']
for t in templates:
    generate('MyApp', 'model.dsmb', t, 'src/MyApp')
```

4. Generate Python Package

```
import subprocess
import base64
import zlib
```

(continues on next page)

(continued from previous page)

```
def generate_package(name, dsmb_path, definitions, output):
    # Generate Python files
    subprocess.run([
        'java', '-jar', 'tools/kibo-1.2.7.jar',
        '-c', 'python',
        '-n', name,
        '-d', dsmb_path,
        '-t', 'templates/python/package',
        '-o', output
    ])

    # Generate embedded definitions
    blob = definitions.encode()
    encoded = base64.b64encode(zlib.compress(blob.encoded()))
    with open(f'{output}/resources.py', 'w') as f:
        f.write(f"B64_DEFINITIONS = {encoded}")
```

Selecting Features

You don't need all templates. Select based on your needs:

Minimal (Data Only)

```
templates = ['Data']  # Just type classes
```

Persistence

```
templates = ['Model', 'Data', 'Attachments', 'Database']
```

Full Stack

```
templates = [
    'Model', 'Data',
    'Stream', 'Json',
    'Attachments', 'Database',
    'ValueType', 'ValueCodec', 'ValueHasher',
    'FunctionPool', 'AttachmentFunctionPool'
]
```

With Testing

```
templates = [..., 'Fuzz', 'Test', 'TestApp']
```

Example: generate.py

A typical project has a `generate.py` script:

```python
#!/usr/bin/env python3
import subprocess
from dsviper import DSMBuilder

NAMESPACE = "MyApp"
DSM_PATH = "definitions/model.dsm"
DSMB_PATH = "build/model.dsmb"
OUTPUT = "src/generated"

# Parse and encode
builder = DSMBuilder.assemble(DSM_PATH)
report, dsm_defs, defs = builder.parse()
with open(DSMB_PATH, "wb") as f:
    f.write(dsm_defs.encode().encoded())

# Generate C++
for template in ['Model', 'Data', 'Attachments', 'Database']:
    subprocess.run([
        'java', '-jar', 'tools/kibo-1.2.7.jar',
        '-c', 'cpp', '-n', NAMESPACE,
        '-d', DSMB_PATH,
        '-t', f'templates/cpp/{template}',
        '-o', OUTPUT
    ])

# Generate Python package
subprocess.run([
    'java', '-jar', 'tools/kibo-1.2.7.jar',
    '-c', 'python', '-n', 'myapp',
    '-d', DSMB_PATH,
    '-t', 'templates/python/package',
    '-o', 'python/myapp'
])

print("Generation complete!")
```

Summary

Step	Tool	Input	Output
Parse	dsviper	`.dsm`	DSMDefinitions
Encode	dsviper	DSMDefinitions	`.dsmb`
Generate	Kibo	`.dsmb` + templates	C++/Python code
Compile	CMake/pip	Generated code	Binary/wheel

Templates let you generate exactly the infrastructure you need, from minimal type definitions to full-stack persistence with RPC.

Getting Started

For your project, copy `exp/generate.py` and keep only the features you need. Or adapt the steps for your build system.

The steps for C++:

1. Save the DSM definitions to a binary representation (`.dsmb`).
2. Generate the resource for the Definitions included by the generated `Model/Definitions.cpp`.
3. Call `kibo-1.2.7.jar` for each feature.

For Python, you can use `dsm_util.py create_python_package` as a shortcut (see *dsm_util*).

Tip

Read the *Template Model Reference* to write your own templates. Study the source code of the provided templates in `templates/cpp/` to learn how to decompose the implementation of a feature.

What's Next

- *Template Model Reference* - Type suffix mechanism, naming conventions, and complete class reference for writing `.stg` templates

3.4.8 Template Model Reference

This reference documents the **Template Model** and **DSM Model** — the two class hierarchies that the StringTemplate engine consumes when generating code from DSM definitions.

Understanding this reference is essential for anyone writing or modifying `.stg` template files.

Code Generation Pipeline

Kibo generates code from a binary representation of DSM definitions. The pipeline is:

1. **Assemble** — Merge multiple `.dsm` files into a single definition set.
2. **Parse** — Produce the AST from the assembled definitions.
3. **Validate** — Check the semantics of the AST.
4. **Encode** — Convert the validated definitions to a binary representation (`.dsmb`).

```
dsm_util.py encode model.dsm model.dsmb
```

5. **Load** — Kibo reads the `.dsmb` file to construct a hierarchy of objects where the root object is an instance of `DSMDefinitions`.
6. **Convert** — The Object-Oriented representation is not suited for the StringTemplate engine, so Kibo converts it to a **Template Model** representation that drastically simplifies feature implementation with templates.

Note

DSM Model classes are **embedded** in the Template Model classes to provide a basic introspection API. Template rules can access both layers.

Type Suffix

During the conversion from DSM Model to Template Model, Kibo recursively collects and decomposes all types and constructs a unique symbol per type called the **type suffix**.

Decomposition Examples

- `float`
  - `_float`
- `vector<float>`
  - `_vector_float`
  - `_float`
- `map<tuple<int64, float>, vector<key<User>>>`
  - `_map_tuple_int64_float_to_vector_UserKey`
  - `_tuple_int64_float`
  - `_int64`
  - `_float`
  - `_vector_UserKey`
  - `_UserKey`

Recursive Template Invocation

The type suffix is used in template rules to call the generated implementation for the inner type (recursion).

In this example, the rule implements the `write` function for the generic type `map<keyType, elementType>`. The `keyTypeSuffix` and `elementTypeSuffix` are used to call the implementation of the `write` function for the `keyType` and for the `elementType`:

```
void StreamWriter::write<v.typeSuffix>(<v.type> const & value) {
    writing->writeUInt64(static_cast<std::uint64_t>(value.size()));
    for (auto const & [k, v] : value) {
        write<v.keyTypeSuffix>(k);
        write<v.elementTypeSuffix>(v);
    }
}
```

The generated code for a `map<int8, string>` is:

```
void StreamWriter::write_map_int8_to_string(std::map<std::int8_t,std::string> const &␣
↪value) {
    writing->writeUInt64(static_cast<std::uint64_t>(value.size()));
    for (auto const & [k, v] : value) {
        write_int8(k);
        write_string(v);
    }
}
```

In theory, any feature can be implemented by recursively consuming the DSM definitions. In practice, it is an art to elaborate the patterns used to implement a templated feature.

Tip

The best way to learn is by studying the templates:

- `templates/cpp/Model/*.stg`
- `templates/cpp/Stream/*.stg`

Template Naming Convention

Type annotations are not available in StringTemplate, so a good parameter naming convention improves understanding of the problem decomposition into template rules.

There is no convention for rule naming. However, the first rule is always `main(m) ::=<<..>>` and the `m` parameter is a `TemplateDefinitions` instance.

Parameter	Usage
`m`	Definitions (module)
`ns`	Namespace
`a`	Attachment
`e`	Enumeration
`em`	Enumeration Member
`s`	Structure
`sf`	Structure Field
`c`	Concept / Club
`cc`	Concept Child
`cd`	Concept Descendant
`cm`	Club Member
`tm`	Tuple Member
`vm`	Variant Member
`po`	Pool
`f`	Function
`p`	Function Parameter
`v`	other

Template Model

During the construction of the template model, the converter substitutes complex types with simple string representations.

For instance, the string values substituted for the type `map<int64, vector<string>>` in an instance of `TemplateMapFunction` are:

```
class TemplateMapFunction {
    //...
    public String type;              // "std::map<std::int64, std::vector<string>>"
    public String typeSuffix         // "_map_int64_to_vector_string"
    public String keyTypeSuffix;     // "_int64"
    public String elementTypeSuffix; // "_vector_string"
    //...
}
```

The following sections use a **pseudo-class** representation to illustrate the fields accessible by the StringTemplate engine to generate code and propagate the recursion.

Note

Since the fields for all Template Model classes are self-explanatory, only the use of the fields for the root class `TemplateDefinitions` is explained in detail to bootstrap the code generation.

TemplateDefinitions

The root class exposes all the DSM definitions of the `model.dsmb` from the Template Model perspective.

```
public class TemplateDefinitions {

    public String generated;
    public String namespace;

    // Namespaces
    public ArrayList<TemplateNameSpace> nameSpaces;

    // Definitions
    public ArrayList<TemplateConcept> concepts;
    public ArrayList<TemplateClub> clubs;
    public ArrayList<TemplateStructure> structures;
    public ArrayList<TemplateStructure> sortedStructures;
    public ArrayList<TemplateEnumeration> enumerations;
    public ArrayList<TemplateAttachment> attachments;
    public ArrayList<TemplateAttachedKeyType> attachedKeyTypes;
    public ArrayList<TemplateAttachedDocumentType> attachedDocumentTypes;

    // Pools
    public ArrayList<TemplateFunctionPool> functionPools;
    public ArrayList<TemplateAttachmentFunctionPool> attachmentFunctionPools;

    // Attachment Pool
    public String attachmentsPoolUuid;

    // Functions
    public ArrayList<TemplateVecFunction> vecFunctions;
    public ArrayList<TemplateMatFunction> matFunctions;
    public ArrayList<TemplateTupleFunction> tupleFunctions;
    public ArrayList<TemplateOptionalFunction> optionalFunctions;
    public ArrayList<TemplateVectorFunction> vectorFunctions;
    public ArrayList<TemplateSetFunction> setFunctions;
    public ArrayList<TemplateMapFunction> mapFunctions;
    public ArrayList<TemplateXArrayFunction> xarrayFunctions;
    public ArrayList<TemplateVariantFunction> variantFunctions;
}
```

The `m` root object is injected in the rule `main(m) ::=<<..>>` and the recursive code generation starts by consuming the fields of the root object:

```
# Anatomy of a complete Feature
# Depending of the Feature, you can omit some fields.

main(m) ::= <<
```

(continues on next page)

(continued from previous page)

```
// Copyright ...
// <m.generated>
...

# generate something by namespace.
<m.nameSpace:namespace(): separator="\n">

# generate something for decomposed types.
<m.vecFunctions:vec(); separator="\n">
<m.matFunctions:mat(); separator="\n">
<m.tupleFunctions:tuple(); separator="\n">
<m.optionalFunctions:optional(); separator="\n">
<m.vectorFunctions:vector(); separator="\n">
<m.setFunctions:set(); separator="\n">
<m.mapFunctions:map(); separator="\n">
<m.xarrayFunctions:xarray(); separator="\n">
<m.variantFunctions:variant(); separator="\n">

>>

namespace(ns) ::= <<
# generate something for enumerations, structures, concepts and clubs.
<ns.concepts:concept(); separator="\n">
<ns.clubs:club(); separator="\n">
<ns.enumerations:enumeration(); separator="\n">
<ns.structures:structure(); separator="\n">

# generate something for attachments.
<ns.attachments:attachments(); separator="\n">
>>
```

TemplateNameSpace

Groups definitions by namespace. Each namespace contains its own concepts, clubs, structures, enumerations and attachments.

```
public class TemplateNameSpace {
    // Namespace
    public String name;

    // Definitions
    public ArrayList<TemplateConcept> concepts;
    public ArrayList<TemplateClub> clubs;
    public ArrayList<TemplateStructure> structures;
    public ArrayList<TemplateStructure> sortedStructures;
    public ArrayList<TemplateEnumeration> enumerations;
    public ArrayList<TemplateAttachment> attachments;
}
```

TemplateConcept

Corresponds to the definition of a `concept`.

```
public class TemplateConcept {
    // DSM
    public DSMConcept dsmConcept;

    // Components
    public TemplateConcept parent;
    public String parentNameInNamespace;

    public ArrayList<TemplateConcept> children;
    public ArrayList<TemplateConcept> descendants;
    public ArrayList<TemplateConcept> strictDescendants;
    public ArrayList<TemplateConceptInNamespace> strictDescendantsInNamespace;

    // Namespace
    public String namespace;
    public String name;

    // Runtime ID
    public String runtimeId;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeSuffix;

    // Attachments
    public ArrayList<TemplateAttachment> attachments;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateConceptInNamespace

Wraps a `TemplateConcept` to provide namespace-qualified names. Used in `strictDescendantsInNamespace` and `membersInNamespace` to generate correct type references when a concept belongs to a different namespace.

```
public class TemplateConceptInNamespace {
    // Component
    public TemplateConcept concept;

    // Namespace
```

(continues on next page)

(continued from previous page)

```
    public String nameInNamespace;
    public String asNameInNamespace;
}
```

TemplateClub

Corresponds to the definition of a `club`.

```
public class TemplateClub {
    // DSM
    public DSMClub dsmClub;

    // Components
    public ArrayList<TemplateConcept> members;
    public ArrayList<TemplateConceptInNamespace> membersInNamespace;
    public ArrayList<TemplateConcept> memberDescendants;

    // Namespace
    public String namespace;
    public String name;

    // Runtime ID
    public String runtimeId;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateEnumeration

Corresponds to the definition of an `enum`.

```
public class TemplateEnumeration {
    // DSM
    public DSMEnumeration dsmEnumeration;

    // Components
    public ArrayList<DSMEnumerationCase> members;
```

(continues on next page)

(continued from previous page)

```
    // Namespace
    public String namespace;
    public String name;

    // Runtime ID
    public String runtimeId;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateStructure

Corresponds to the definition of a `struct`.

```
public class TemplateStructure {
    // DSM
    public DSMStructure dsmStructure;

    // Components
    public ArrayList<TemplateStructureField> fields;

    // Predicates
    public boolean isMovable;

    // Namespace
    public String namespace;
    public String name;

    // Runtime ID
    public String runtimeId;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeSuffix;
```

(continues on next page)

(continued from previous page)

```
    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateStructureField

```
public class TemplateStructureField {
    // DSM
    public DSMStructureField dsmField;

    public String name;
    public String passBy;
    public String defaultValue;

    // Predicates
    public boolean isMovable;
    public boolean isTypeAny;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeInNamespace;
    public String typeSuffix;

    // Field
    public TemplateField field;

    // Viper
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateAttachment

Corresponds to the definition of an `attachment`.

```
public class TemplateAttachment {
    // DSM
    public String dsmAttachment;

    public String representation;
    public String identifier;
```

(continues on next page)

(continued from previous page)

```
    // Namespace
    public String namespace;
    public String name;

    // Runtime ID
    public String runtimeId;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public TemplateAttachedKeyType keyType;
    public TemplateAttachedDocumentType documentType;

    // Python
    public String pythonIdentifier;
}
```

TemplateAttachedKeyType

```
public class TemplateAttachedKeyType {
    // Namespace
    public String namespace;
    public String name;

    // Type
    public String type;
    public String typeInNamespace;
    public String typeSuffix;

    // Viper
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateAttachedDocumentType

```
public class TemplateAttachedDocumentType {
    // Predicates
    public Boolean isStructure;
    public Boolean useBlobId;

    // Component
    public TemplateStructure structure;

    // Type
```

(continues on next page)

(continued from previous page)

```
    public String type;
    public String typeInNamespace;
    public String typeSuffix;

    // Field
    public TemplateField field;

    // Viper
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateField

This class is used to generate **path-based mutators** per container.

```
public class TemplateField {
    public String passBy;

    // Predicates
    public Boolean isNotBox;
    public Boolean isBox;
    public Boolean isSet;
    public Boolean isMap;
    public Boolean isXArray;

    // Type
    public String type;
    public String keyType;
    public String keyTypeSuffix;
    public String elementType;
    public String elementTypeSuffix;

    // Viper
    public String elementTypeViperValue;

    // Python
    public TemplatePythonType pythonKeyType;
    public TemplatePythonType pythonElementType;
}
```

TemplateFunctionPool

Corresponds to the definition of a `function_pool`.

```
public class TemplateFunctionPool {
    // DSM
    public String name;
    public String uuid;
```

(continues on next page)

(continued from previous page)

```
    // Components
    public ArrayList<TemplateFunction> functions;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
}
```

TemplateFunction

```
public class TemplateFunction {
    // DSM
    public String name;

    // Components
    public ArrayList<TemplateFunctionParameter> parameters;

    // Predicates
    public Boolean isVoid;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeSuffix;

    // Viper
    public String returnViperValue;

    // Python
    public TemplatePythonType returnPythonType;
}
```

TemplateFunctionParameter

```
public class TemplateFunctionParameter {
    // DSM
    public String name;

    public String passBy;

    // Type
    public String type;
    public String typeSuffix;
```

(continues on next page)

(continued from previous page)

```
    // Viper
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
}
```

TemplateAttachmentFunctionPool

Corresponds to the definition of an `attachment_function_pool`.

```
public class TemplateAttachmentFunctionPool {
    // DSM
    public String name;
    public String uuid;

    // Components
    public ArrayList<TemplateAttachmentFunction> functions;

    // Documentation
    public boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
}
```

TemplateAttachmentFunction

```
public class TemplateAttachmentFunction {
    // DSM
    public String name;

    // Components
    public ArrayList<TemplateFunctionParameter> parameters;

    // Predicates
    public boolean isVoid;

    // Documentation
    public Boolean hasDocumentation;
    public String documentation;

    // Type
    public String type;
    public String typeSuffix;

    // Type
    public String interfaceType;
```

(continues on next page)

(continued from previous page)

```
    // Viper
    public String returnViperValue;

    // Python
    public TemplatePythonType returnPythonType;
}
```

TemplateVecFunction

Corresponds to the definition of a `vec<elementType, size>`.

```
public class TemplateVecFunction {
    // DSM
    public String dsmType;
    public String size;

    // Type
    public String type;
    public String typeSuffix;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
    public TemplatePythonType getPythonElementType;
    public String pythonTupleType;
}
```

TemplateMatFunction

Corresponds to the definition of a `mat<elementType, columns, rows>`.

```
public class TemplateMatFunction {
    // DSM
    public String dsmType;
    public String columns;
    public String rows;

    // Type
    public String type;
    public String typeSuffix;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
```

(continues on next page)

(continued from previous page)

```
    public TemplatePythonType pythonType;
    public TemplatePythonType pythonElementType;
    public String pythonTupleType;
    public String pythonColumnType;
}
```

TemplateType

Represents a generic type member used in `tuple` and `variant` definitions.

```
public class TemplateType {
    // Type
    public String type;
    public String typeSuffix;
}
```

TemplateTupleFunction

Corresponds to the definition of a `tuple<T0, ...>`.

```
public class TemplateTupleFunction {
    // DSM
    public String dsmType;

    // Components
    public ArrayList<TemplateType> members;

    // Type
    public String type;
    public String typeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
    public ArrayList<TemplatePythonType> pythonMembers;
}
```

TemplateOptionalFunction

Corresponds to the definition of an `optional<elementType>`.

```
public class TemplateOptionalFunction {
    // DSM
    public String dsmType;

    // Type
    public String type;
    public String typeSuffix;
```

(continues on next page)

(continued from previous page)

```
    public String elementType;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
    public TemplatePythonType pythonElementType;
}
```

TemplateVectorFunction

Corresponds to the definition of a `vector<elementType>`.

```
public class TemplateVectorFunction {
    // DSM
    public String dsmType;

    public String valueRef;

    // Type
    public String type;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
    public TemplatePythonType pythonElementType;
}
```

TemplateSetFunction

Corresponds to the definition of a `set<elementType>`.

```
public class TemplateSetFunction {
    // DSM
    public String dsmType;

    // Type
    public String type;
    public String typeSuffix;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;
```

(continues on next page)

(continued from previous page)

```
    // Python
    public TemplatePythonType pythonType;
    public TemplatePythonType pythonElementType;
}
```

TemplateMapFunction

Corresponds to the definition of a `map<keyType, elementType>`.

```
public class TemplateMapFunction {
    // DSM
    public String dsmType;

    // Type
    public String type;
    public String typeSuffix;
    public String keyTypeSuffix;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
    public TemplatePythonType pythonKeyType;
    public TemplatePythonType pythonElementType;
}
```

TemplateVariantFunction

Corresponds to the definition of a `variant<T0, ...>`.

```
public class TemplateVariantFunction {
    // DSM
    public String dsmType;

    // Components
    public ArrayList<TemplateType> members;

    // Type
    public String type;
    public String typeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
```

(continues on next page)

(continued from previous page)

```
    public ArrayList<TemplatePythonType> pythonMembers;
}
```

TemplateXArrayFunction

Corresponds to the definition of a `xarray<elementType>`.

```
public class TemplateXArrayFunction {
    // DSM
    public String dsmType;

    // Type
    public String type;
    public String typeSuffix;
    public String elementTypeSuffix;

    // Viper
    public String viperType;
    public String viperValue;

    // Python
    public TemplatePythonType pythonType;
    public TemplatePythonType pythonElementType;
}
```

TemplatePythonType

Used to generate Python proxy classes for Viper.

```
public class TemplatePythonType {
    // Proxy
    public Boolean useProxy;
    public String proxy;

    // Type
    public String typeSuffix;
    public String type;
}
```

DSM Model

The DSM Model is the projection of the DSM definitions into a hierarchy of Java classes.

The pseudo-classes below express the public API that the StringTemplate engine consumes when evaluating rule substitutions.

Note

These classes are **embedded** in the Template Model classes for basic introspection. For example, `TemplateConcept.dsmConcept` gives access to the underlying `DSMConcept`.

DSMDefinitions

The root class references all the DSM definitions found in the `model.dsmb`.

```
public class DSMDefinitions {
    public ArrayList<DSMConcept> concepts;
    public ArrayList<DSMClub> clubs;
    public ArrayList<DSMEnumeration> enumerations;
    public ArrayList<DSMStructure> structures;
    public ArrayList<DSMAttachment> attachments;

    public ArrayList<DSMFunctionPool> functionPools;
    public ArrayList<DSMAttachmentFunctionPool> attachmentFunctionPools;
}
```

DSMTypeReference

Used when a reference to another type is needed. For example, to express the list of members of a `club`.

```
public class DSMTypeReference {
    public TypeName typeName;
    public DSMTypeReferenceDomain domain;
}
```

DSMConcept

Corresponds to the definition of a `concept`.

```
public class DSMConcept {
    public TypeName typeName;
    public DSMTypeReference parent;
    public String documentation;
    public DSMTypeReference typeReference;
    public UUID runtimeId;
}
```

DSMClub

Corresponds to the definition of a `club` with related `membership`.

```
public class DSMClub {
    public TypeName typeName;
    public ArrayList<DSMTypeReference> members;
    public String documentation;
    public DSMTypeReference typeReference;
    public UUID runtimeId;
}
```

DSMEnumeration

Corresponds to the definition of an `enum`.

```
public class DSMEnumeration {
    public TypeName typeName;
    public ArrayList<DSMEnumerationCase> members;
    public String documentation;
    public DSMTypeReference typeReference;
    public UUID runtimeId;
}

public class DSMEnumerationCase {
    public String name;
    public String documentation;
}
```

DSMStructure

Corresponds to the definition of a `struct` and provides access to the definition of the structure fields.

```
public class DSMStructure {
    public TypeName typeName;
    public ArrayList<DSMStructureField> fields;
    public String documentation;
    public DSMTypeReference typeReference;
    public UUID runtimeId;
}

public class DSMStructureField {
    public String name;
    public DSMType type;
    public DSMLiteral defaultValue;
    public String documentation;
}
```

DSMLiteral

Used to express the literal value for the initialization of a field.

```
public abstract class DSMLiteral {
    public abstract String representation();
}

public class DSMLiteralValue extends DSMLiteral {
    public DSMLiteralDomain domain;
    public String value;
}

public final class DSMLiteralList extends DSMLiteral {
    public final ArrayList<DSMLiteral> members;
}
```

DSMType

The base class of types.

```
public abstract class DSMType {
    public abstract String representation();
}
```

DSMTypeVec

Corresponds to the definition of a `vec<T, n>`.

```
public class DSMTypeVec extends DSMType {
    public DSMTypeReference elementType;
    public long size;
}
```

DSMTypeMat

Corresponds to the definition of a `mat<T, columns, rows>`.

```
public class DSMTypeMat extends DSMType {
    public DSMTypeReference elementType;
    public long columns;
    public long rows;
}
```

DSMTypeTuple

Corresponds to the definition of a `tuple<T0, ...>`.

```
public class DSMTypeTuple extends DSMType {
    public ArrayList<DSMType> types;
}
```

DSMTypeOptional

Corresponds to the definition of an `optional<T>`.

```
public class DSMTypeOptional extends DSMType {
    public DSMType elementType;
}
```

DSMTypeVector

Corresponds to the definition of a `vector<T>`.

```
public class DSMTypeVector extends DSMType {
    public DSMType elementType;
}
```

DSMTypeSet

Corresponds to the definition of a `set<T>`.

```
public class DSMTypeSet extends DSMType {
    public DSMType elementType;
}
```

DSMTypeMap

Corresponds to the definition of a `map<K, V>`.

```
public class DSMTypeMap extends DSMType {
    public DSMType keyType;
    public DSMType elementType;
}
```

DSMTypeVariant

Corresponds to the definition of a `variant<T0, ...>`.

```
public class DSMTypeVariant extends DSMType {
    public ArrayList<DSMType> types;
}
```

DSMTypeXArray

Corresponds to the definition of a `xarray<elementType>`.

```
public class DSMTypeXArray extends DSMType {
    public DSMType elementType;
}
```

DSMFunctionPool

Corresponds to the definition of a `function_pool`.

```
public class DSMFunctionPool {
    public UUID uuid;
    public String name;
    public ArrayList<DSMFunction> functions;
    public String documentation;
}

public class DSMFunction {
    public DSMFunctionPrototype prototype;
    public String documentation;
}

public class DSMFunctionPrototype {
    public String name;
    public ArrayList<DSMFunctionPrototypeParameter> parameters;
```

(continues on next page)

(continued from previous page)

```
    public DSMType returnType;
}

public class DSMFunctionPrototypeParameter {
    public String name;
    public DSMType type;
}
```

DSMAttachmentFunctionPool

Corresponds to the definition of an `attachment_function_pool`.

```
public class DSMAttachmentFunctionPool {
    public UUID uuid;
    public String name;
    public ArrayList<DSMAttachmentFunction> functions;
    public String documentation;
}

public class DSMAttachmentFunction {
    public boolean isMutable;
    public DSMFunctionPrototype prototype;
    public String documentation;
}
```

CHAPTER

FOUR

INDICES

- genindex - All classes and methods
- search - Full-text search

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z