

Table of Contents

-  Purpose
-  Design Pattern
-  Features
- Data Processing Flow
-  Testing Philosophy
-  License
-  Usage
-  Related Crates

 Core library for modeling, validating, and transforming CSDIF data.

Purpose

This crate provides the reusable core logic for working with CSDIF data structures. It is designed to be used by multiple adapters, including:

- `csdif-cli` — command-line exporter
- `csdif-api` — web-based API service

It implements the format and validation layer of the CSDIF RFC, but does not include any I/O, CLI, or web-specific logic.

Design Pattern

This crate represents the **core hexagon** in a hexagonal architecture (Ports & Adapters). It contains:

- Domain models (`Sensor` , `Observation` , `Location` , `CsdifDocument`)
- Validation logic (`validate_document`)
- Transformation and mapping functions

Adapters such as CLI or web-API interact with this crate via public interfaces, but this crate remains unaware of its consumers.

Features

- ✔ CSDIF-compliant data structures
- ✔ JSON serialization and deserialization
- ✔ Validation of document structure and semantics
- ✔ Mapping and transformation helpers
- ✗ No I/O, CLI, or async dependencies

Data Processing Flow

Incoming CS-DIF data passes through the following stages:

Step	Description	Responsible Component	Extensibility
1. Transport (Inbound)	Data arrives via HTTP, MQTT, File, etc.	<code>csdif-api</code> , <code>csdif-mqtt</code> , <code>csdif-cli</code>	Add new transport modules
2. Format Decoding	Raw data is decoded from JSON, XML, CSV, etc.	<code>csdif-core::format</code>	Implement new <code>FormatDecoder</code> traits
3. Schema Interpretation	Data is interpreted according to the initiative-specific schema	<code>csdif-mapper-*</code>	Implement <code>InitiativeMapper</code> trait
4. Mapping to SensorML	Data is converted into the internal <code>SensorML</code> domain model	<code>csdif-mapper-*</code> + <code>sensorml</code>	Add new mappers per initiative
5. Validation & Processing	<code>SensorML</code> document is validated and optionally enriched	<code>sensorml</code>	Extend domain validation via traits
6. Mapping to External Schema (optional)	<code>SensorML</code> is transformed into a target schema if needed	<code>csdif-core::mapper::*</code>	Implement <code>SchemaExporter</code> trait
7. Format Encoding	Data is encoded into JSON, XML, CSV, etc.	<code>csdif-core::format</code>	Implement <code>FormatEncoder</code> traits

	...		
8. Transport (Outbound)	Data is sent via HTTP, MQTT, File, etc.	csdif-api, csdif-mqtt, csdif-cli	Add new transport modules

This layered architecture ensures that new formats, transports, or initiative-specific schemas can be added without modifying the core logic. Each layer is pluggable and testable in isolation.

Testing Philosophy

- All logic is covered by unit tests
- No `unwrap`, `?`, or `panic!` allowed
- Errors are handled explicitly and surfaced clearly
- JSON roundtrip tests ensure format stability
- Validation logic is tested with both valid and invalid cases

License

All source files include:

```
// SPDX-License-Identifier: MIT
```

RUST

Usage

Add to your `Cargo.toml` :

```
csdif-core = { git = "https://github.com/your-org/csdif-core", branch = "main" }
```

TOML

Or use crates.io once published:

```
csdif-core = "0.1.0"
```

TOML

Then in your code:

```
use csdif_core::{CsdifDocument, validate_document};
```

RUST

Related Crates

- `csdif-cli` : CLI tool for exporting and validating CSDIF documents
- `csdif-api` : Web service exposing CSDIF validation and transformation endpoints