Schema2ReST Project Specification

**Schema2ReST** is a project designed to streamline the process of creating RESTful APIs from high-level schema definitions. It uses an enhanced schema description format based on Mermaid Model Diagrams (MMD) with custom decorators to describe data entities, validations, inheritance, unique constraints, and relationships. This enhanced schema is then converted to YAML and subsequently used to generate both data models and REST API routers automatically.

Table of Contents

[Schema2ReST Project Specification 1](#_Toc190331452)

[1. Introduction 2](#_Toc190331453)

[2. Overview 2](#_Toc190331454)

[3. The Process 3](#_Toc190331455)

[1.1 3](#_Toc190331456)

[4. MMD Enhancements 3](#_Toc190331457)

[4.1 Inheritance 3](#_Toc190331458)

[3.2 Unique Constraints 4](#_Toc190331459)

[3.4 Dictionaries 4](#_Toc190331460)

[3.5 Validations 5](#_Toc190331461)

[5. Relationships 5](#_Toc190331462)

[6. MongoDB Dependency 6](#_Toc190331463)

[7. Model and Router Generation 7](#_Toc190331464)

[6.1 Model Generator (gen\_model) 7](#_Toc190331465)

[6.2 Router Generator (gen\_router) 7](#_Toc190331466)

[8. Run time environment 7](#_Toc190331467)

[9. Testing and Integration 8](#_Toc190331468)

[10. Summary and Future Enhancements 8](#_Toc190331469)

# Overview

The primary goal of Schema2ReST is to allow developers to define their entire data schema in a single, concise format and automatically generate the underlying code for:

* **Models:** Pydantic/Beanie models that represent data objects in a MongoDB database, including all field definitions, validations, inheritance, and unique constraint enforcement.
* **Routers:** REST API endpoints that rely on the generated models to enforce consistent validation and business logic.

The system is tightly integrated with MongoDB as its back-end database. It uses the [Beanie ODM](https://github.com/roman-right/beanie) for MongoDB, which provides Mongo-specific types (such as PydanticObjectId) and asynchronous query methods. This Mongo dependency influences several aspects of the project, from field type definitions to query logic for enforcing unique constraints.

The project consists of these major components:

* **Enhanced MMD Input:** A custom schema description language that extends standard Mermaid MMD by adding decorators for validations, inheritance, unique constraints, and relationships.
* **MMD-to-YAML Converter:** A converter that parses the Enhanced MMD file and produces a structured YAML file containing the full schema.
* **Model Generator (gen\_model):** A generator that reads the YAML file and creates Python model code. The generated models:
  + Merge field definitions with validation constraints.
  + Support inheritance by extending a BaseEntity (which contains Mongo-specific fields and behaviors).
  + Enforce unique constraints by overriding the save() method to perform runtime checks.
* **Router Generator (gen\_router):** A generator that creates REST API endpoint code. Since the unique constraint enforcement is handled at the model level, no changes are required in the router generation logic.

# The Process

A Makefile is supplied that takes a schema and creates a running ReST server. Following are the specific steps:

## Generate a yaml file

## Create Base system files

Main.py

Db.py

# MMD Enhancements

The following snippet of the MMD includes enhances as describe herein. All enhancements are noted by a comment (%%) followed by a decorator (@).

There are 4 types of decorators – Inheritance, Uniques, Dictionaries and Validators

erDiagram

BaseEntity {

ObjectId \_id

ISODate createdAt

ISODate updatedAt

%% @validate \_id: { required: true, autoGenerate: true }

%% @validate createdAt: { required: true, autoGenerate: true }

%% @validate updatedAt: { required: true, autoUpdate: true }

}

Account {

ISODate expiredAt

%% @inherit BaseEntity

%% @validate expiredAt: { required: false }

}

## Inheritance

* **Inheritance Decorator:**  
  Entities can specify inheritance using a decorator such as:

%% @inherits BaseEntity

This indicates that the entity inherits from a BaseEntity, which provides common fields (like \_id, createdAt, and updatedAt) and Mongo-specific behavior. The YAML output includes an "inherits" key that is a list of parent entity names.

@inherits must be defined within an entity

## Unique Constraints

* **Unique Decorator:**  
  Unique constraints can be defined via statements like:

%% @unique lastname

%% @unique email + username

Each unique statement specifies a set of fields that must be unique. In the example above, lastname must be unique and email combined with username must be unique.

* **YAML Representation:**  
  In the YAML, unique constraints are represented by a key "uniques" whose value is a list of dictionaries. Each dictionary contains a key "fields" mapping to a list of field names (e.g., {"fields": ["email"]} or {"fields": ["email", "username"]}).
* **Model Enforcement:**  
  The generated model code includes a method (validate\_uniques()) that checks for duplicate records with the same field values before saving. If a duplicate is found, the model raises a custom exception (e.g., UniqueValidationError) with details about the violation.

@uniques must be defined within an entity

## Dictionaries

Dictionaries can be defined and are global in scope. Dictionaries have a name and a set of key/values as follows:

%% @dictionary pattern {

email: "^[a-zA-Z0-9.\_%-][+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$](mailto:+@[a-zA-Z0-9.-]+\.%5ba-zA-Z%5d%7b2,%7d$)",

url: "^https?://[^\s]+$"

}

**Note**: Dictionary statements must be defined on a single physical line. It is broken up above for clarity. The same dictionary name can be used to extend a dictionary.

The above defines the dictionary named “pattern” which has 2 keys – email and url. These are shortcuts that are used as validation values using the format dictionary=pattern.email

Dictionaries are defined outside of any entity definition

## Validations

* **Validation:**  
  The Enhanced MMD syntax uses explicit markers to define each validation. Within an entity definition, validations are specified by a series of validation lines, each following the format:

%% @validate url: { required: true, pattern: dictionary=pattern.url, pattern.message: "Bad URL format" }

%% @validate recurrence: { required: false, enum: [daily, weekly, monthly, yearly] }

Validation attributes include:

* + required (true/false)
  + minLength and maxLength (numeric limits)
  + pattern (regex patterns, which must be enclosed in quotes if containing colons or spaces)
  + enum (a list of allowed values, typically provided as a string that can be parsed into a list)

A validation attribute may also be in the form <attribute>.message as in pattern and pattern.message shown above. There are default messages for every attribute, however certain attributes (pattern, minLength, maxLength, enum) can have an optional .message which will be used to override the default message.

Validation values can be a simple value, an enum, a regex pattern string or a dictionary lookup. In the above, the url pattern gets its value from the url key in the pattern dictionary. Alternatively, patterns can be direct regex values – no dictionary lookup required.

* **Merging Logic:**  
  The MMD-to-YAML converter merges the validations into the entity’s field definitions so that each field’s output includes its type along with its validation constraints.

# Relationships

* **Relationship Parsing:**  
  Relationships between entities are defined in the MMD and processed into:
  + A top-level \_relationships list in the YAML (a list of dictionaries with keys "source" and "target", where "source" is the child entity and "target" is the parent).
  + Each entity also has a "relations" array that lists the parent entities for that entity.

For example:

yaml

Copy

\_relationships:

- source: Account

target: User

- source: UserEvent

target: User

- source: UserEvent

target: Event

# MongoDB Dependency

Schema2ReST is built on top of MongoDB using the Beanie ODM. This introduces several Mongo-specific aspects:

* **Data Model Types:**  
  Models inherit from Beanie’s Document class. Fields such as \_id are defined as PydanticObjectId, a type that represents MongoDB ObjectIds.
* **Default Factories and Timestamps:**  
  The BaseEntity model automatically generates an \_id using a default factory and manages timestamps (createdAt and updatedAt) using Mongo-specific behavior.
* **Asynchronous Operations:**  
  Database operations (such as saving documents and querying for unique constraints) are performed asynchronously using MongoDB’s async drivers as wrapped by Beanie.
* **Query Logic:**  
  The unique constraint logic in the generated models uses Mongo-style queries (e.g., await self.\_\_class\_\_.find\_one(query)) to enforce uniqueness.

Switching to a different back-end would require re-implementing these parts, as the current design is tightly coupled with MongoDB.

# Model and Router Generation

## 6.1 Model Generator (gen\_model)

* **Merging Fields and Validations:**  
  The generator reads the YAML schema and merges the field definitions with their corresponding validations. It converts string representations of booleans and numbers appropriately and converts enum strings into lists.
* **Inheritance:**  
  The generator uses the "inherits" key to extend the appropriate base class. For example, if an entity inherits from BaseEntity, the generated model imports BaseEntity and extends it.
* **Unique Constraints:**  
  If unique constraints are defined (via the "uniques" key), the generator produces additional logic in the model:
  + A custom exception class (UniqueValidationError) is generated.
  + An asynchronous method validate\_uniques() is created that, for each unique constraint, builds a query using the specified fields and checks the database for existing records. If a record is found, UniqueValidationError is raised.
  + The model’s save() method is overridden to call validate\_uniques() before saving.

## 6.2 Router Generator (gen\_router)

* **Minimal Changes:**  
  Since unique constraints are enforced in the model’s save() method, the router generator does not need significant changes. The routers simply call the model methods, and any unique constraint violations are propagated as errors from the model layer.

# Run time environment

There is a config.json file in the project root directory that includes the Mongo connection, database name and other relevant information as follows:

{

"mongo\_uri": "mongodb://localhost:27017",

"db\_name": "eventMgr",

"host": "127.0.0.1",

"app\_port": 5500,

"log\_level": "info",

"environment": "development"

}

This data is loaded at runtime to establish the running environment

# Testing and Integration

* **REST API Testing:**  
  The generated REST API (e.g., using FastAPI) automatically converts incoming JSON into the proper types using Pydantic. The integration of models with the REST API ensures that all validations (including unique constraint checks) are applied.
* **Direct Model Testing:**  
  Unit tests can directly call model methods (using pytest and pytest-asyncio) to verify that validations, uniqueness, and inheritance behave as expected.
* **Interactive Testing:**  
  With VSCode’s Python extension and integrated pytest support, you can run and debug both model-level and API-level tests.

# Summary and Future Enhancements

**Schema2ReST** provides a single source of truth for defining your data schema using an enhanced MMD syntax. It supports:

* **Field Validations:** Merging constraints such as required status, length limits, regex patterns, and allowed enumerations.
* **Inheritance:** Allowing entities to inherit common fields and behavior from a BaseEntity.
* **Unique Constraints:** Enforcing uniqueness through custom model logic and raising detailed errors when constraints are violated.
* **Relationships:** Capturing entity relationships both in a top-level \_relationships key and within each entity’s "relations" array.

Because the project is built on MongoDB using Beanie, it leverages Mongo-specific data types and asynchronous operations. This tight integration with MongoDB simplifies development for that back-end, although moving to a different persistence layer would require significant rework of the model and query logic.

Future enhancements might include support for additional validation types, more flexible back-end abstraction, and improved error reporting/logging