Design Patterns

# Selected Patterns

## Observer:

* From the system requirements:
  + *Assessment clashing: The system should monitor all assessments being added to the calendar and determine which clashes are permitted and which should be avoided. When an avoidable clash is encountered, the staff members are notified, and the assessment is sent to the admin for approval for it to be accepted or rejected.*

## Builder

* Removes the large number of constructor parameters featured in some classes (e.g., Staff and CourseAssessment), replacing them with method calls to an appropriate class implementing the Builder interface.
  + Particularly useful for CourseAssessment, which has mostly nullable attributes.

## Singleton

* Only need one instance of each builder.

## State

* From the constraints:
  + *When a clash is encountered involving a Group 1 assessment, the assessment is not scheduled but rather, the schedule request is sent to the admin for approval. Once approved, it is set. If rejected, the staff members attached to such a course are notified and required to reschedule the assessment.*

## Repository

* Separates data(base) access logic from data structure logic (i.e., models) and business logic (i.e., controllers).
  + Alternatively, abstracts the data access from the rest of the application.
* Used to simplify CRUD (create, read, update, delete) operations.
* ***Single-responsibility principle.***

## ~~Serializer~~

* ~~Separates data~~ *~~representation~~* ~~logic (e.g., to\_json(), to\_csv()) from data structure logic (i.e., models).~~
* ***~~Single-responsibility principle.~~***

## Factory

* Dynamically creates the necessary repositories and serializers.

# Sources

* [What Is The Repository Pattern And How To Use It In Python? - Pybites](https://pybit.es/articles/repository-pattern-in-python/)
* [The Factory and Repository Pattern with SQLAlchemy and Pydantic | by Lawson Taylor | Medium](https://medium.com/@lawsontaylor/the-factory-and-repository-pattern-with-sqlalchemy-and-pydantic-33cea9ae14e0)

# ChatGPT on MVCSR:

## Query:

what are the responsibilities of the model, repository, service, controller, and view, and how are they ordered from lowest level to highest level?

## Response:

In a typical Model-View-Controller-Service-Repository (MVCSR) architecture, each layer has distinct responsibilities, which help organize code, separate concerns, and improve scalability and maintainability. Here's a breakdown of each component, their responsibilities, and their order from the lowest level (closest to the database) to the highest level (closest to the user).

**1. Model (Lowest Level)**

* **Responsibilities**:
  + Defines the structure of data and the rules governing it.
  + Represents the data objects that are mapped to database tables or documents (e.g., a User model for a users table).
  + Implements data-related logic, such as validation rules and default values.
  + Sometimes includes methods specific to the model, like relationships to other models (e.g., a User having a relationship with Orders).
* **Typical Content**:
  + Attributes defining data fields.
  + Methods for basic operations on the data (e.g., set\_password, check\_password).
* **Level**: **Lowest** – Closest to the database, abstracting raw data into meaningful entities.

**2. Repository**

* **Responsibilities**:
  + Manages data access and performs database operations.
  + Encapsulates all code that interacts with the database (queries, inserts, updates, deletes).
  + Abstracts the persistence layer (database) and offers a higher-level API for working with the data.
  + Provides methods like find\_by\_id, find\_all, or more specialized queries (e.g., find\_user\_by\_email).
* **Typical Content**:
  + CRUD operations for specific models, sometimes specialized for certain use cases.
  + Uses models to access the data but abstracts away low-level database logic from the rest of the application.
* **Level**: Second-Lowest – Interfaces with the model but abstracts direct database access for higher layers.

**3. Service**

* **Responsibilities**:
  + Contains business logic, which is the core of application behavior.
  + Orchestrates operations that might involve multiple models or repositories.
  + Provides methods like register\_user, authenticate\_user, or process\_order, which involve more than simple CRUD operations.
  + Can perform complex logic, such as sending notifications, validating business rules, or managing transactions.
* **Typical Content**:
  + High-level methods implementing application logic.
  + Calls repository methods to fetch or store data, manipulates models, and returns processed information.
* **Level**: Mid-Level – Interfaces with repositories and models, providing higher-level methods to the controllers.

**4. Controller**

* **Responsibilities**:
  + Manages HTTP requests from the client and directs them to the appropriate service methods.
  + Processes input from the request, such as URL parameters, JSON payloads, or form data.
  + Calls service methods to retrieve data or perform operations and handles the response.
  + Converts service responses to a format appropriate for the client (e.g., JSON for REST APIs).
  + Typically, minimal logic – its main role is coordination.
* **Typical Content**:
  + Methods that handle specific routes, e.g., login\_user, register\_user, or get\_user\_by\_id.
  + Calls service methods and decides on HTTP response codes.
* **Level**: Second-Highest – Interfaces with the service layer and serves as an intermediary between services and views.

**5. View (Highest Level)**

* **Responsibilities**:
  + Presents data to the user, typically in the form of HTML, JSON, or XML.
  + In web applications, the view can be an HTML template rendered with data passed from the controller.
  + For API responses, the view may be omitted or handled within the controller, which directly sends JSON or XML.
  + Decouples the presentation logic from the business logic, allowing different representations of the same data.
* **Typical Content**:
  + HTML templates (for server-rendered applications).
  + JSON structure definitions or serializers if using a REST API.
* **Level**: **Highest** – Interfaces directly with the end-user, translating data and responses into a user-consumable format.

**Ordering from Lowest to Highest Level**

**1. Model** ⟶ **2. Repository** ⟶ **3. Service** ⟶ **4. Controller** ⟶ **5. View**

This ordering follows a pattern of abstraction:

* **Models** provide raw data structures and relationships.
* **Repositories** handle database interactions and provide access to models.
* **Services** encapsulate business logic and coordinate repositories.
* **Controllers** handle client requests, orchestrating service calls and responses.
* **Views** provide a formatted response to the user or client application, consuming data from the controller layer.

**Example: User Authentication Flow**

Let's walk through an example to illustrate the responsibilities:

1. **User Model**: Defines attributes like id, email, and password\_hash.
2. **User Repository**: Provides methods like find\_user\_by\_email(email), which queries the database for a user by email.
3. **User Service**: Defines the authenticate\_user(email, password) method, which:
   * Retrieves the user from the repository.
   * Verifies the password and returns the user if authentication succeeds.
4. **User Controller**: Exposes an endpoint for /login:
   * Calls authenticate\_user in the service.
   * Returns a success response with user data or an error message.
5. **User View**: (API-based) Returns the JSON response to the client app:
   * { "user\_id": 123, "message": "Login successful" }.

* Each layer has a clearly defined role, with only minimal overlap, making the codebase modular, testable, and maintainable.