SPRING Boot

Best Practices for Projects

musanabiyev

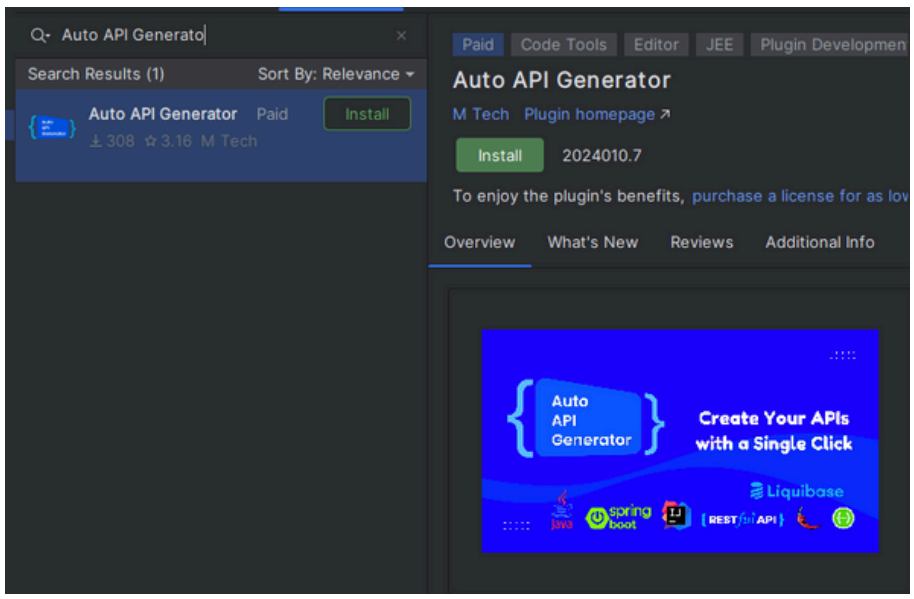
musanabiyev

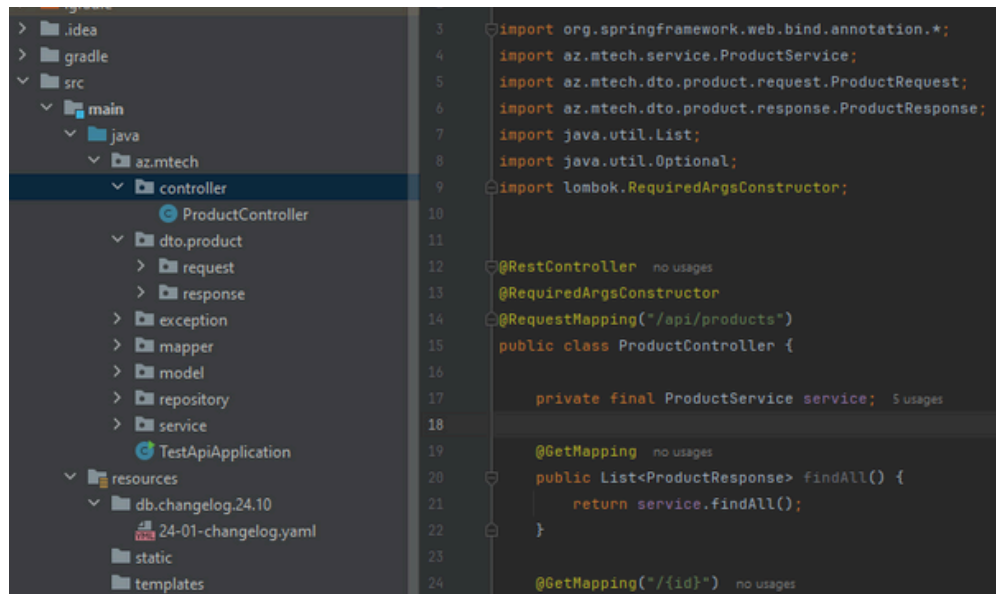
Spring Boot is a widely used framework that simplifies the development of applications within the Java ecosystem. It enables the rapid creation of complex applications with minimal configuration. It is popular for building microservices and RESTful APIs.

Key advantages of Spring Boot:

- **Embedded servers** (Tomcat, Jetty) – No need to install a separate server.
- **Auto-Configuration** – Most of the configuration is done automatically.
- **Production-ready features** – Includes functionalities like monitoring, metrics, and health checks.

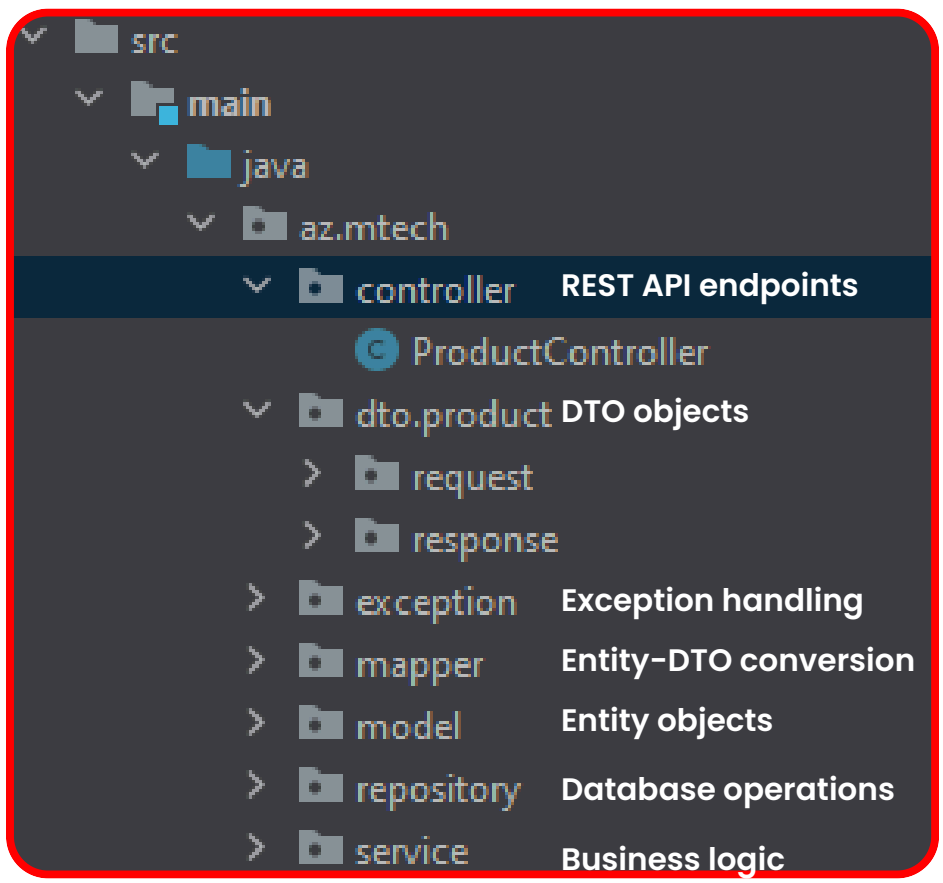
Apply best practices with our Auto API Generator plugin based on your entity class! In seconds, it automatically sets up packages, classes, code, and functionalities. Your structure will be fully automated, eliminating the need for manual setup!





With our Auto API Generator plugin, you can complete in minutes what would normally take a month! Speed up your work, save time, and focus on more important projects. Auto API Generator is the perfect solution to boost your development speed!

Project Architecture



Project Workflow

1. The **user** sends a request to the /api/products endpoint.
2. The **Controller** receives the request and forwards it to the **Service** Layer.
3. The **Service** Layer executes the business logic, retrieves the data from the **Repository**, and converts the **Entity** to a **DTO** using the **Mapper**.
4. The **Repository** Layer interacts with the database and returns the result.
5. The **Mapper** prepares the response by converting the result from **Entity** to **DTO**.
6. If everything is successful, the **Controller** sends the response to the user.
7. In case of an error, the **Exception Handler** manages the error and returns an appropriate message.

1. Controller (API Layer)

- This folder contains the REST API endpoints.
- Marked with the @RestController annotation, it handles HTTP requests (GET, POST, PUT, DELETE).
- It calls the Service Layer and returns the results to the user in *JSON* format.

Examples:

- /products endpoint returns a list of products.
- /products/{id} endpoint displays a specific product.

2. Service Layer (Business Logic)

- This layer handles the business rules and logic.
- Processes the data received from the **Controller**.
- All dependencies are injected using constructor injection.

Example:

- When adding a new product, stock and price limitations are checked here.

3. Repository Layer (Database Operations)

- Establishes a connection with the database.
- Provides CRUD operations through **JpaRepository** or **CrudRepository**.

Example:

- **ProductRepository** is used to query product objects from the database.

4. Model Layer (Entity and DTO Objects)

- This layer stores both **Entity** and **DTO** classes.
- Entity objects correspond to the database tables.
- **DTO** objects are used for data transfer and prevent direct sharing of Entities with the API.

Examples:

- **Product Entity**
- **ProductDTO** – an object that returns only the necessary information.

5. Mapper Layer (Entity-DTO Conversion)

- This layer handles the conversion between **Entity** and **DTO**.
- Can utilize **MapStruct** or manual mapper classes.

Example:

- The **ProductMapper** class converts a **Product** object to a **ProductDTO**.

6. Exception Layer (Error Handling)

- This layer contains specific exception classes and a **Global Exception Handler** to manage errors occurring in the project.
- Global error management is ensured with the @ControllerAdvice annotation.

Examples:

- **ProductNotFoundException** – thrown when a product is not found.
- **GlobalExceptionHandler** – returns a standard response for all errors occurring in the project.

Here are some best practices to apply when using the application.yml file:

1. **Organized Structure:** Group configurations hierarchically (e.g., datasource, server, security).
2. **Profile Management:** Create profiles for different environments (development, test, production).
3. **Environment Variables:** Use environment variables instead of hardcoding sensitive information directly.
4. **Default Values:** Define default values for configurations.
5. **Clear Comments:** Add comments to clarify configuration fields.
6. **Grouping:** Collect related configurations in one place.
7. **Custom Configuration:** Include application-specific parameters and create custom configuration classes.

Here are some additional components you can add to your project:

1. **Security:** Implement user authentication and authorization with Spring Security.
2. **Configuration:** Create a separate configuration class for API keys and database parameters.
3. **Validation:** Add the @Valid annotation to your DTOs for validation.
4. **Logging:** Use SLF4J or Logback for logging.
5. **Caching:** Cache frequently used data with Spring Cache or Redis.
6. **Testing:** Write tests using JUnit and Mockito.
7. **Monitoring:** Monitor your application's health with Spring Boot Actuator.
8. **Event Handling:** Implement a system to manage events.
9. **Asynchronous Processing:** Use the @Async annotation to handle long-running operations asynchronously.
10. **Documentation:** Document your APIs using Swagger.

This structure ensures that the code is clear and maintainable over the long term. Each component has its own responsibilities and interacts minimally with other layers. With this approach, you can more easily extend and maintain your project.

musanabiyev musanabiyev