!! USE INTERFACES AND SEPARATE CLASSES FOR STORING DATA !!

!! The data storage implementation should be selected with profiles !!

!! Use proper OOP and write tests !!

!! EACH TASK SHOULD ON *GITHUB* AND ITS OWN BRANCH AND/OR PULL REQUEST !!

Step 1: Basic REST API with In-Memory Storage

- Learn basic differences between Gradle and Maven
- Create a Spring Boot project using start.spring.io.
- Implement a simple `TaskController` with endpoints (`GET`, `POST`, `DELETE`).
- Implement a simple `UserController` with endpoints (`GET`, `POST`).
- Implement a simple `NotificationController` with endpoints (2x `GET`).
- Implement separate classes for each controller and service!
- Store tasks in a `List` or `Map` (no database yet!!)
- Return JSON responses with adequate http codes.

Step 2: Write unit-tests

- Using JUnit or TestNG write unit-tests for your application

Step 3: In-Memory Database (H2)

- Add **H2 database** as an in-memory database.
- Configure Spring Data JPA.
- Convert in-memory `HashMap` storage to a database-backed repository.
- Implement `Repository` for all services using Spring Data JPA.

Step 4: Add Docker Support

- Write a `Dockerfile` for the Spring Boot application.
- Use **Docker Compose** to start the database and the app together.
- Test the application running in containers.

Step 5: Switch to a database(PostgreSQL, MongoDB, Cassandra, InfluxDB, Firebase, Clickhouse…)

- Replace H2 with **PostgreSQL**.
- Update `application.properties` for PostgreSQL connection.
- Use **Flyway** for database migrations.
- Write new tests. Use mockito to mock responses from the database.

Step 6: Implement Caching (Redis, Valkey, Dragonfly, Memcached or any other, but consult with your teacher!)

- Use **Spring Cache**.
- Cache task retrieval to improve performance.
- **Search for entries in Caching database, and if not found, then search in database**
- **Set timeouts for values**

Step 7: Implement Messaging (RabbitMQ, Kafka, Artemis, Pulsar, ActiveMQ, NATS…)

- Set up **RabbitMQ** or **Kafka** or any other message broker. (RabbitMQ is a bit simpler, Kafka is faster)
- Publish a message when a new task is created.
- Remake the Notification service to receive updates !! ONLY !! from the message broker.
- Create a listener to process messages asynchronously.

Step 8: Add Scheduling & Async Tasks

- Use **@Scheduled** to periodically check for overdue tasks.
- Use **@Async** for background processing.

Step 9: Split Monolit into microservices

- Enable **Spring Boot Actuator** for health checks and metrics.
- Integrate **Prometheus & Grafana** for monitoring.
- Update docker-compose

Step *10*: Rewrite the Tasks service using Webflux

- Use R2DBC
- Make a integration/stress test and compare speed