A system for temporary job placement services

Lab1 – Setting up a basic service

Learning objectives

- Use GitHub as a repository and a collaboration tool
- Create and deploy a simple web application
- Check input parameters
- Manage runtime exceptions and map them to error descriptions
- Test API through an HTTP client

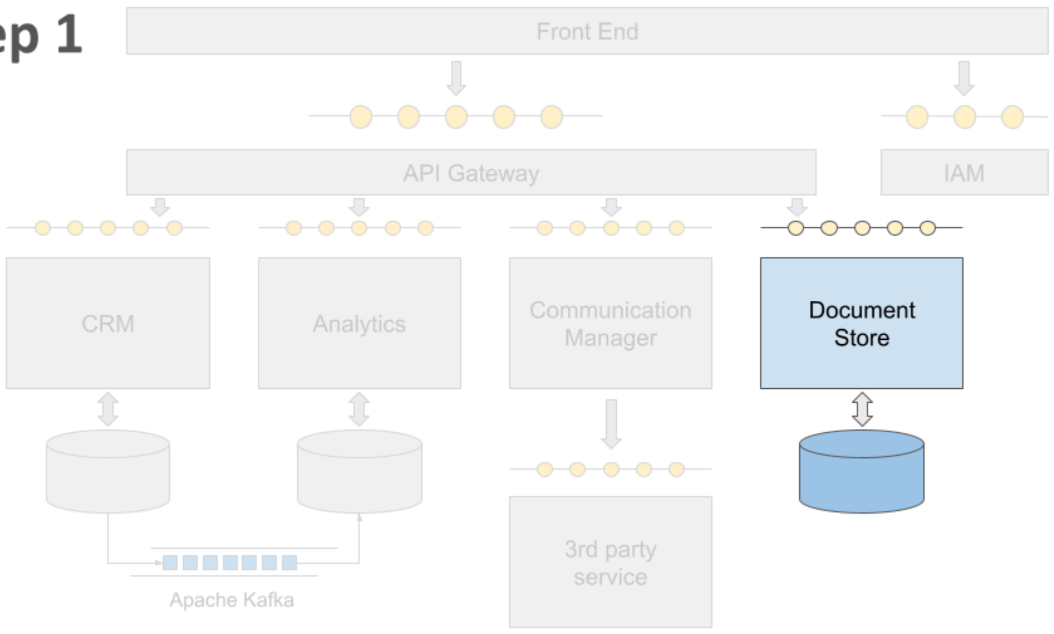
Description

A company provides temporary job placement services. It relies on a Customer Relationships Management tool to streamline its operations as well as improve its effectiveness and efficiency.
This tool acts as a centralized platform to manage and enhance interactions with candidates, clients, and other stakeholders involved in the hiring process by providing a centralized hub for managing candidates, job offers, and client relationships. It helps recruiters and HR professionals to work more effectively, make data-driven decisions, and ultimately improve the overall quality of placements.

One of the fundamental functionalities provided by this system is document management. To fulfill this objective, the creation of a specialized component is necessary, aiming to:

1. Centralize storage for contracts, resumes, and other important documents
2. Track document changes within the CRM system

Step 1



This component, which will subsequently evolve into a microservice within the comprehensive system, is designated for storing *binary data*. Its development adheres to a **tiered approach**, necessitating the establishment of a client tier, a web tier, and a data tier.

To realize the client tier, for this first lab, you don't need a front-end application. You can use a simple HTTP client, like the one integrated into IntelliJ IDEA, or any application to test your API (e.g., Postman).

The data tier is represented by a database and you will use *PostgresDB* running inside a *Docker* container.

Similarly, the web/server tier is organized into separate **layers**. Through the utilization of SpringBoot, the implementation involves establishing a presentation layer, a service layer, and a data access layer.

The service should provide **CRUD** operations to help users manage their documents effectively. This means that clients should be able to upload new documents, find and download specific files, retrieve a comprehensive list of files, update existing files with new versions, and delete any documents that are no longer required.

The objective of this initial laboratory is to create a fundamental service, while simultaneously gaining insight into how to utilize the Spring Framework, Gradle, Kotlin, and Docker effectively to achieve this goal.

The laboratory submissions will be done by pushing code inside a GitHub private repository provided to you through the GitHub Classroom system.

Prerequisites

- IntelliJ IDEA
- Docker desktop / docker engine and docker compose

Steps

1. Set up your profile inside GitHub Classroom. This must be done only once, accepting the first group assignment for Lab1.

    a. Load page: https://classroom.github.com/a/7qgRBJqw

    b. Login with your GitHub account (or create a new one) and authorize GitHub Classroom system to access your profile

    c. Associate your GitHub account with your user identity in the full list that appears.
    Note: the prefix of each identity is the group identifier chosen in https://docs.google.com/spreadsheets/d/1ARsKb47XMk1wJkf5cDVjwOotytpgRQ1zQ1i7ZISZKk8/edit#gid=0

    d. Then you must join a group. If you are the first person in your group to complete the procedure, please create a new team with the **same group name** specified in the Google Sheet. If one of your colleagues has already created your team, just join it. Pay attention to not joining the **wrong team**

    e. Now, a new repository for lab1 group assignment has been created, and you can start collaborating with your colleagues using it. Take care to manage push operations, and not overwrite existing code. Consider using a well-known methodology like Git Flow (https://infinum.com/handbook/android/building-quality-apps/using-git)

    f. You can fork a new branch with the aim of developing features and also use release/tag strategy to mark milestones during the development

    g. The deadline of the assignment is a cutoff date: after the date is reached, you will lose write access to the repository

    h. The **last commit** before the deadline or the version of the project tagged as "completed" will be considered for evaluation

2. Run IntelliJ IDEA and select File → New → Project from Version Control…: Select the repository URL in the dialog box and proceed.
3. The repository will contain only one module: a Spring Boot web application named “**document_store**”.
    a. Create the module via File → New → Module…: choose Spring Initializr, name it “document_store”, opt for the Kotlin language and the Gradle build tool. In the next screen choose Spring Boot version 3.2.4 and select, as dependencies:
        i. Spring Web
        ii. Spring Data JPA
        iii. PostgreSQL Driver
        iv. Docker Compose Support
    b. Commit following the Conventional Commits specification: https://www.conventionalcommits.org/en/v1.0.0/
    Then push the commit to the GitHub repository.
4. The *spring-boot-docker-compose* module creates automatically a new file called *compose.yaml* in the project's root folder, with similar content.

```
services:
  postgres:
   image: 'postgres:latest'
   environment:
      - 'POSTGRES_DB=mydatabase'    #optional
      - 'POSTGRES_PASSWORD=secret' #optional
      - 'POSTGRES_USER=myuser'     #optional
   ports:
      - '5432'
```

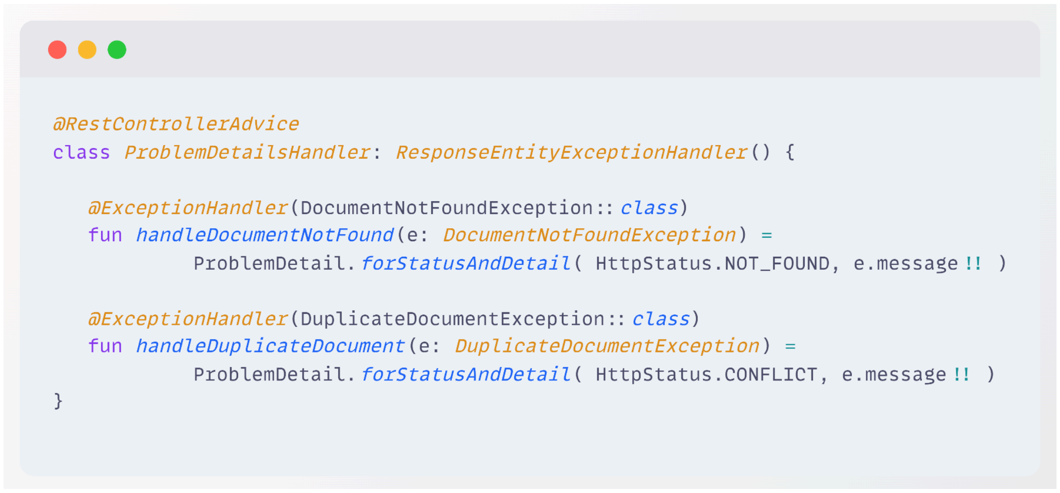
5. Since Spring Boot 3.1, starting the application the framework will detect that there's a Docker Compose file present, and will run the command *docker compose up* for you before connecting to the services. If the services are already running, it will detect that, too, and will use them. It will also run *docker compose stop* when the application shuts down.
    a. Containers can take some time to become fully ready. The framework ensures that the application starts only when the other services are ready.
6. The provided *compose.yaml* file creates a service named "**postgres**" based on the "postgres:latest" image, utilizing container port 5432. The host port is chosen randomly to prevent conflicts with other running containers.

   a. The database is ready to accept connections from the application.

7. In the **document_store** module, edit the *src/resources/application.properties* file and set the "spring.jpa.show-sql" key to true and "spring.jpa.hibernate.ddl-auto" to "update".
   a. The database connection URL entry is unnecessary since the *spring-boot-docker-compose* dependency will automatically detect the correct port and inject it for us, following the schema [*jdbc:postgresql://host:port/database?properties*].
8. Implement the **document_store** module, managing REST requests targeted to the following endpoints:
   a. GET /API/documents/ -- list all registered documents in the DB. Consider using page number and limit as request parameters
   b. GET /API/documents/{metadatatId}/ -- details of document {documentId} or fail if it does not exist
   c. GET /API/documents/{metadatatId}/data/ -- byte content of document {metatadataId} or fail if it does not exist
   d. POST /API/documents/ -- convert the request param into a DocumentMetadataDTO and store it in the DB, provided that a file with that name doesn't already exist. Use multipart/form-data as content-type and MultiPartFile as type for your request parameter. Additionally, it is important to log the changes made to the file at *info* level.
   e. PUT /API/documents/{metadataId} -- convert the request param into a DocumentMetadataDTO and replace the corresponding entry in the DB; fail if the file does not exist.
      i. It is necessary to update also the content of the file itself, not only the metadata associated with it. Additionally, it is important to log the changes made to the file at *info* level.
   f. DELETE /API/documents/{metadataId} -- remove document {documentId} or fail if it does not exist. Additionally, it is important to log the changes made to the file at *info* level.
9. Consider the structure of the document and its storage within a database. A document is represented by its binary content along with pertinent metadata: name, size, content type, and creation timestamp. It's advisable to create two separate repositories, each corresponding to a distinct entity class, interconnected through a one-to-one relationship.

10. All endpoints should return a valid JSON object and a corresponding HTTP status code.

11. The application code should throw custom subclasses of RuntimeException to notify of any error it may detect. To manage these exceptions, add a class similar to the following one:



```kotlin
@RestControllerAdvice
class ProblemDetailsHandler: ResponseEntityExceptionHandler() {

   @ExceptionHandler(DocumentNotFoundException::class)
   fun handleDocumentNotFound(e: DocumentNotFoundException) =
            ProblemDetail.forStatusAndDetail( HttpStatus.NOT_FOUND, e.message!! )

   @ExceptionHandler(DuplicateDocumentException::class)
   fun handleDuplicateDocument(e: DuplicateDocumentException) =
            ProblemDetail.forStatusAndDetail( HttpStatus.CONFLICT, e.message!! )
}
```

    a. Remember to set *spring.mvc.problemdetails.enabled* to **true** in the *application.properties* file to enable *problem detail* standard

12. Commit code and push it to the repository.

13. Write a set of HTTP messages to verify the exposed endpoints

    a. Test each one individually, with well-formed request and invalid request

    b. Build a chain of requests to simulate the expected usage of this software component.

        i. e.g., client **requests all** the documents → client **uploads** a new document → client **checks** if the document is now present with a new request → client **downloads** the document → client **deletes** the document

14. (Optional in this phase) By default, all files created inside a container are stored on a writable container layer. This means the data doesn't persist when that container no longer exists.
Use [docker volumes]() to persist data managed by **document_store** service without losing it when you perform a *docker compose down*

    a. Pay attention to the correct usage of *spring.jpa.hibernate.ddl-auto* application property.

Submission rules

- The work must be submitted by April, 11 23:59
- The last commit before the deadline will be evaluated. Alternatively, create a release and label it as “completed”.