Technische Universität München
Fakultät für Informatik

Exercise Sheet 3: DSM, Modular Design, and Spring Boot Service

Welcome to the third week of Advanced Topics of Software Engineering. This week, we will focus on the concepts of dependency structure matrix and guidelines for modular design which you have learned from the lectures. First, we are going to examine the dependency structure matrix of a Java framework and interpret how optimal it is. Later, we will practice modular design guidelines such as coupling & cohesion and Liskov's substitution principle.

This week, we start with developing REST API with Spring Boot, Maven and MongoDB. Do not be intimidated by the terminologies during the system development, you will get to know them step-by-step.

Theoretical Exercise

Exercise 1: Dependency Structure Matrix

Dependency Structure Matrix (DSM) provides a compact way to represent relationships among elements of a software system. Using DSM, a software developer can get a better understanding of components that belong together. This becomes handy when it comes to *partitioning* an application and analyzing design concerns such as *cohesion* and *coupling*.

DSMs can be generated with most modern IDEs on module, package or class level. In the following, we analyze the DSM of the Spring Boot package, generated with IntelliJ IDEA.

1. Have a look at the package view of the DSM in Fig. 1. How often does the package `org.springframework.boot.web` use other packages' classes? Note that `...` means there are more than 100 uses.
2. How often do other packages use classes from `org.springframework.boot.web`?
3. How does an optimal DSM looks? How can one spot cyclical dependencies in a DSM? Are there any cyclic dependencies in `org.springframework.boot`?
4. With respect to cohesion and coupling, what insights can you gain from inspecting a row or column of the DSM? Is there a package which can suggests an existence of a **God object** in `org.springframework.boot`?

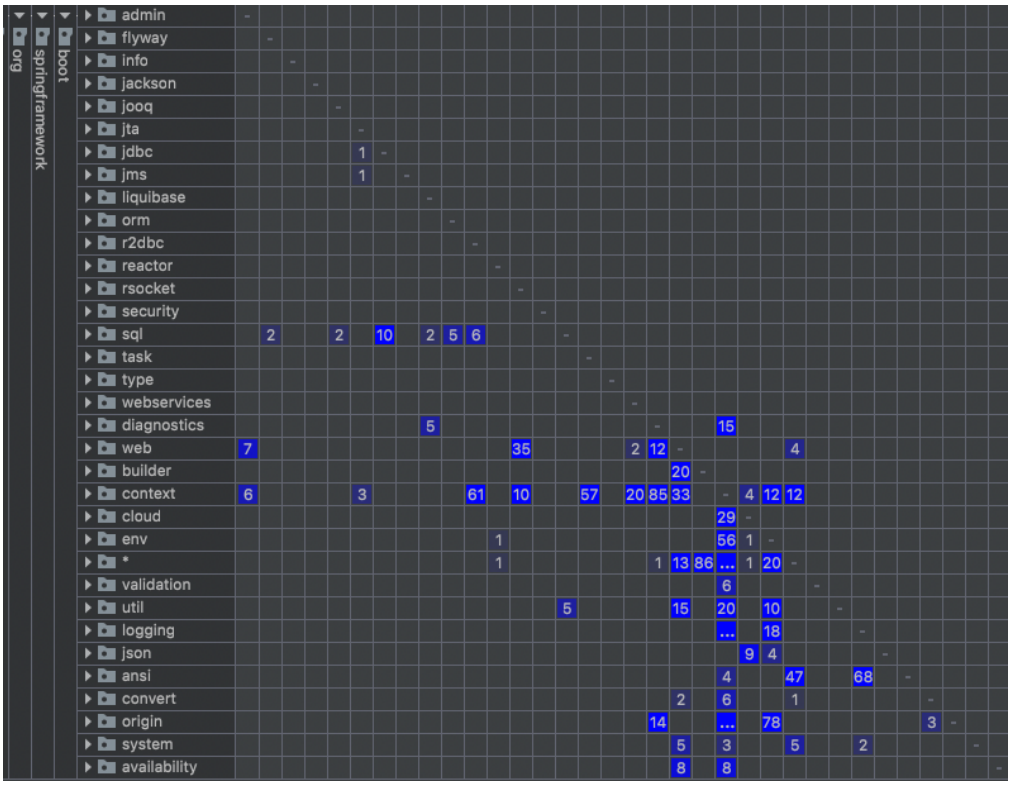

Figure 1: DSM for `org.springframework.boot`

Exercise 2: Software Architecture Principles

In the lecture, we have seen different software architecture best practices and guidelines for *good* modular design. In this exercise, you will assess a simple architecture according to the introduced quality attributes and principles.

The *EZCalculator* is a brand-new calculator to revolutionize the math education in junior high schools. Its key selling points are its simplicity and extensibility, as well as its basic and advanced math mode.

Below you can see the simplified high-level architecture of the *EZCalculator*.

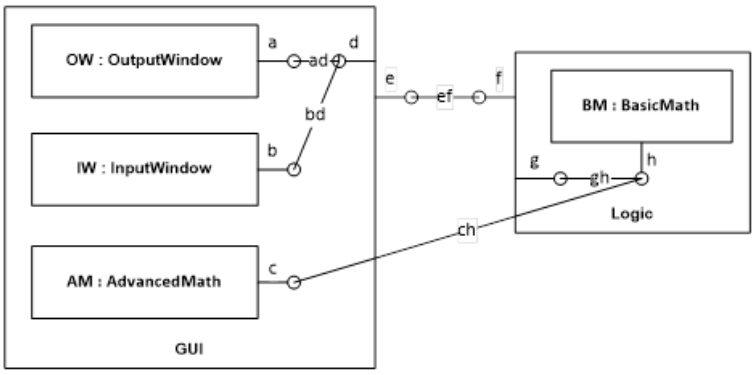

Figure 2: High Level Architecture of the *EZCalculator*

1. Given the architecture of the *EZCalculator* depicted in Fig. 2, formally represent the structure of the *EZCalculator* according to the formal definition of the structure of a system (the *system model* $S := (C, I, CON)$) introduced in the lecture.
2. Continue with your system model. Calculate the coupling factor of the system ($\alpha = 0.5$). What are the two summands of the coupling factor? How can you interpret them w.r.t. the degree of coupling in the system? What is the factor α good for?
3. Based on the coupling factor and observations in the high level architecture, do you think this system

is *well* decoupled? Can you think of any improvements? Do these improvements affect the coupling factor?

4. Considering the semantics of the different *EZCalculator* components, given by the respective names, do you say the functional cohesion of the system is rather low or high? Why? If low, how can you fix it? How would this fix affect the decoupling of the system?

5. In conclusion, which architectural principles does the default architecture of the *EZCalculator* violate?

Exercise 3: Liskov's Substitution Principle

As you already know, Liskov's Substitution Principle (LSP) states that if a program module is using a Base class T, then an object of type T can be replaced with an object of a derived class (i.e. subclass) S without affecting the functionality of the program module.

Rent-a-vehicle is a company offering their services to customers who need to temporarily rent a vehicle in order to transport people or goods on land. The company provides a brochure to their customers, where they can check all vehicles' features:

```
public abstract class Vehicle {
  private Door door1;
  private Door door2;
  private Door door3;
  private Door door4;

  public abstract void drive();
  public abstract void addLuggage();
  public abstract void playRadio();

  public void unlockFourDoors() {
    door1.unlock();
    door2.unlock();
    door3.unlock();
    door4.unlock();
  }
}
```

Additionally, the company policy regarding renting a vehicle is that the customers do not choose the vehicle themselves. On the contrary, they get the next available vehicle:

```
public class VehicleHireService {
  // ...
  public Vehicle hireVehicle() {
    return availableVehiclePool.getNextVehicle();
  }
}
```

Consider the following situation: You are currently on a holiday in a foreign country and want to rent a vehicle for you and your family.

1. Imagine, on delivery day, you get a two-seat Lamborghini sports car. Create a class `LamborghiniCar` that extends the Vehicle class including a reasonable constructor (remember: abstract classes cannot be instantiated, but they can be subclassed).

2. Does your new `LamborghiniCar` class violate the LSP? Justify.

3. Now, imagine get a BMW X1 Series SUV (four-seat) car on delivery day (instead of the sports car). Create a class `BMWX1Car` that extends the `Vehicle` class by a reasonable constructor.

4. Does your new `BMWX1Car` class violate the LSP? Justify.

Practical Exercise

This exercise sheet will introduce Maven, a build management tool. Additionally, you will learn how to create REST endpoints to receive requests, MongoDB database entities, and repositories for querying a MongoDB database using the Spring Boot Framework.

Your goal this week should be to **deepen your understanding of Spring Boot**, as we will use it extensively throughout the next weeks to demonstrate advanced concepts.

Exercise 4: Prerequisites/Setup

1. You have an IDE (e.g., IntelliJ IDEA[1]) installed or some other development setup.
2. You have Git installed.
3. You have a Java development kit (JDK) and runtime environment (JRE) installed and configured properly (we recommend to use at least version 1.8)[2].
4. Additionally you need to install Apache Maven from https://maven.apache.org/. While the installation for Maven is straightforward on macOS and Linux, Windows users might find this tutorial helpful: http://www.mkyong.com/maven/how-to-install-maven-in-windows/.
5. We suggest you use Maven's command line interface (CLI). This requires the Maven `bin` directory to be part of your PATH environment variable (https://en.wikipedia.org/wiki/PATH_(variable)). Although we recommend other alternatives, feel free to use Window's default `cmd.exe` command prompt.

Ultimately, ensure your setup is working by typing `mvn -v` into the command line. The command should provide you version information.

Exercise 5: Maven

`Maven` (https://maven.apache.org/), similar to `ant` or `make`, is a build system. In contrast to these tools it uses conventions for the build procedure. The advantage is that all maven projects will have a common structure and only the exceptions to the conventions need to be specified. Another selling point of maven is that it allows the developer to specify dependencies in the build file. This makes it easier to manage large projects that depend on a lot of different libraries or plugins. `Gradle`, another build management tool, is one of the most popular alternatives to Maven, which we will get to know later in the course.

Take a look at the following example maven build file (`pom.xml`) (usually called POM for Project Object Model in the Maven jargon):

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
       >
4    <modelVersion>4.0.0</modelVersion>
5
6    <groupId>edu.tum.ase</groupId>
7    <artifactId>mavenstarter</artifactId>
8    <version>0.0.1-SNAPSHOT</version>
9    <packaging>jar</packaging>
10
11   <name>mavenstarter</name>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15     <maven.compiler.target>1.8</maven.compiler.target>
16     <maven.compiler.source>1.8</maven.compiler.source>
17   </properties>
18
```

[1] IntelliJ IDEA: https://www.jetbrains.com/idea
[2] OpenJDK: https://openjdk.java.net

```
19    <dependencies>
20      <dependency>
21        <groupId>junit</groupId>
22        <artifactId>junit</artifactId>
23        <version>4.13</version>
24        <scope>test</scope>
25      </dependency>
26    </dependencies>
27  </project>
```

Answer the following questions by going through the official Maven documentation and guide[3].

1. How do you have to change this POM to, for instance, include the *Bouncy Castle* encryption library in your project?
2. How could you add additional repositories to this POM (an example could be the *Restlet* module[4])?
3. Where in the file system do you have to create the source file for the class `edu.tum.ase.mavenstarter.App`?
4. Which maven command do you need to use to generate a `.jar` file?
5. How can you execute your packaged application with the generated `.jar` file?

Exercise 6: Configure an Authorized MongoDB Database Connection in Spring

For your delivery system, one of the services that you will offer is a *box service*, which has a connection to the box collection inside a MongoDB database. This collection stores and queries boxes as documents. However, in this exercise, we will not "spoil" your creativity by directly developing a box service. Instead, we will introduce the concept to you in a different context, so you can apply it to design your ASE Delivery system.

We will work with a project management system, where each project contains the name and source code files. These projects can still be retrieved when the users return to the application, refresh the web page, or allow authorized dispatchers to delete entities that they do not need anymore.

We use a non-relational database (MongoDB) for the project, but this is only one way for data persistence.

The following exercises lead you through the process of creating a so-called **Spring Bean**, which handles the Object-Document Mapping (ODM) to a Collection in a MongoDB database. Beans are components defined at the core of the Spring framework. A bean defines a singleton[5] inside a Spring application and can be injected into other beans by Spring's dependency injection mechanism[6]. Basically, all classes annotated with one of Spring's internal component annotators such as `@Controller`, `@RestController`, `@Service`, `@Component`, or `@Repository` are beans. In Spring Boot, these are automatically registered as Spring Beans when you start your application due to the `@SpringBootApplication` annotation, which we already saw in the practical exercise, as it enables an automatic scan for classes annotated with internal component annotators [7]. To inject a registered bean into another bean, you can simply annotate the injected bean with `@Autowired`.

By the end of these exercises, you will have a skeleton of the project service for the ASE Project implemented. The procedure of the upcoming exercises is described in Fig. 3:

1. Setup an authorized connection to a MongoDB Database
2. Define a Document model of a `Project` as a Java class

[3] Maven in 5 Minutes: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html
[4] Restlet Maven Setup: https://restlet.com/open-source/documentation/user-guide/2.3/introduction/getting-started/maven
[5] Singleton pattern: https://en.wikipedia.org/wiki/Singleton_pattern
[6] Dependency injection in Spring: https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-spring-beans-and-dependency-injection.html
[7] Spring Boot Application annotation: https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html

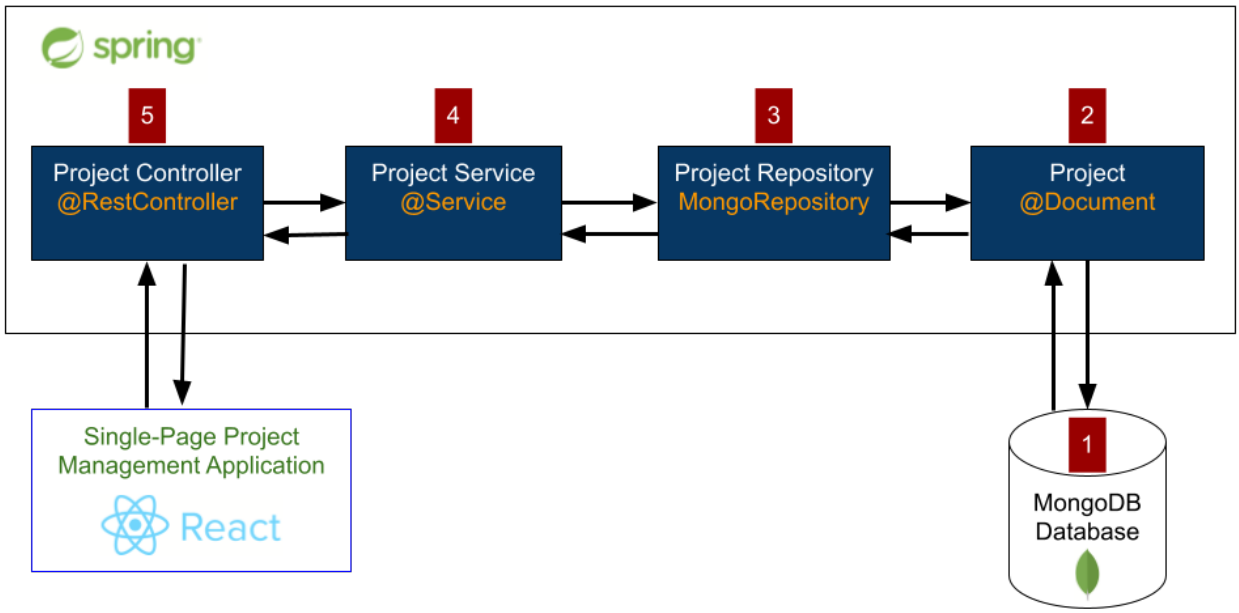

Figure 3: C4 Component Diagram of the *project service*

3. Create a Repository to execute database operations and store the operation result using the `Project` Document
4. Develop a Service to provide interactions with `Project` via the Repository
5. Setup a controller to receive REST requests and call the corresponding service for managing projects

It entails a MongoDB ODM to store and get a list of persisted projects by using the Spring Data MongoDB API through Spring's convenient wrappers.

As outlined before, we will need to configure a connection to a database which persists projects and their source code files created and shown within the OnlineIDE.

Therefore, it is straightforward to create a singleton (i.e. Spring Bean) responsible for holding properties of the database connection (e.g. username, password, database name, or address) which we will do in the following.

The *Java Persistence API* (JPA) is a standard technology that lets you "map" objects to documents in non-relational databases like MongoDB, or entities in relational databases, such as MySQL, PostgreSQL.

In order to establish a connection to a collection in a database with Spring, we will use the *Spring Data MongoDB* project.

- Use the Spring Initializr (https://start.spring.io/) to generate and download a new Spring Boot Project with the following options:
  - Project: Maven
  - Language: Java
  - Spring Boot: 2.5.4 (or the latest)

  - Group: edu.tum.ase
  - Artifact: project
  - Packaging: jar
  - Java: 8 (or higher)
  - Dependencies: Spring Web, Spring Data MongoDB

- Put the **content** of the `.zip` file into a sub-directory `project` in your repository
- Import the project into your IDE as a Maven project
- Start a MongoDB database server running on port 27017 either by first installing MongoDB from the official website (https://www.mongodb.com/try/download/community), select the right OS platform that you use to develop your system. Next, install the *mongosh* https://www.mongodb.com/try/download/shell?jmp=docs as the MongoDB shell for interaction with database. Additionally, if you prefer a GUI interface to interact with MongoDB, you can download MongoDB Compass from https://www.mongodb.com/try/download/community.

  If you are familiar with *Docker*, by running it in a Docker container (you must have Docker installed on your system) with the command [8]:

  ```
  docker run -d --name project-mongo -e MONGO_INITDB_ROOT_USERNAME=aseAdmin -e
      MONGO_INITDB_ROOT_PASSWORD=test mongo
  ```

  Do not worry if you are not familiar with Docker yet, we will cover it in future exercises in more depth.

- If you are not using Docker and the command above, create a database user `aseAdmin` with password `test` in the default authentication database called `admin`.

  You can either use the MongoDB Shell *mongosh* to do this. [9]

  Note that by default, MongoDB database does not have a default user and password, so we need to enable user authentication in MongoDB. This folder resides in */etc/mongod.conf* in Linux, */usr/local/etc/mongod.conf* or */opt/homebrew/etc/mongod.conf* in macOS, or *[install directory]\bin \mongod.cfg* in Windows. Change the `security` config into:

  ```
  security:
          authorization: "enabled"
  ```

- Connect to the database first using the following command:

  ```
  mongosh --port 27017
  ```

- Next, switch to the *admin* database, and create the `aseAdmin` user.

  ```
  use admin

  db.createUser(
    {
      user: "aseAdmin",
      pwd: passwordPrompt(), // or cleartext password
      roles: [
        { role: "userAdminAnyDatabase", db: "admin" },
        { role: "readWriteAnyDatabase", db: "admin" }
      ]
    }
  )
  ```

- Exit the *mongosh* by typing `exit()`. Next, connect to MongoDB via *mongosh* again with the credential:

  ```
  mongosh --port 27017  --authenticationDatabase "admin" -u "aseAdmin" -p
  ```

- Spring has an application configuration file located at `src/main/resources/application.properties`. Add the following database configuration into the `application.yml` file:

[8] MongoDB in Docker: https://hub.docker.com/_/mongo
[9] Create a User Admin in MongoDB: https://docs.mongodb.com/manual/tutorial/enable-authentication/#connect-to-the-instance

```yaml
    # Configuration for the connection string to MongoDB database
    spring:
      data:
        mongodb:

          # Configure database location
          host: localhost
          port: 27017
          database: aseProject

          # Configure credential
          username: aseAdmin
          password: ase
          authentication-database: admin

          # Configure Database Operation,
          # allow creating index automatically from the code
          auto-index-creation: true

          #enable the usage of Spring Data Mongo Repository
          repositories:
            enabled: true
```

- You may wonder, we specify the `aseProject` as the target database, but we have not created it yet. Can Spring throw an error at this stage? We will see the magic afterward, and not only for MongoDB Database.
- If you configured everything properly and your MongoDB server is running on the correct port (as defined in `application.properties`), you should be able to start the Spring application which will per default run on port 27017. Remember you can run the Spring application either from the command line by using `./mvnw spring-boot:run` or through your IDE by selecting and running the `ProjectApplication.java` class where the `main()` method is defined.
- What happened behind the scenes is that Spring created a `com.mongodb.MongoClient` bean and configured it with the set properties. Let's inject it into another bean and see if we can print details about it to the console. We only have the class `ProjectApplication.java` defined so far, which is inherently a Spring bean as well (through the `@SpringBootApplication` annotation). As aforementioned, we can inject beans into other beans by using `@Autowired`. Additionally, we utilize `CommandLineRunner`, which is a simple Spring Boot interface with a `run` method. Spring Boot will automatically call the run method of all beans implementing this interface after the application context has been loaded.

```java
package edu.tum.ase.project;

import com.mongodb.client.MongoClient;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;

@SpringBootApplication
public class ProjectApplication implements CommandLineRunner {

  private static final Logger log = LoggerFactory.getLogger(ProjectApplication.class);

  @Autowired
  MongoClient mongoClient;

  public static void main(String[] args) {
    SpringApplication.run(ProjectApplication.class, args);
  }

  @Override
  public void run(String... args) throws Exception {
    log.info("MongoClient = " + mongoClient.getClusterDescription());
  }
}
```

- When running the application again you should see something like the following in your output console:

```
2021-09-22 18:08:53.765  INFO 17036 --- edu.tum.ase.project.ProjectApplication    :
    MongoClient = ClusterDescription{type=STANDALONE, connectionMode=SINGLE,
    serverDescriptions=[ServerDescription{address=localhost:27017, type=STANDALONE, state=
    CONNECTED, ok=true, minWireVersion=0, maxWireVersion=8, maxDocumentSize=16777216,
    logicalSessionTimeoutMinutes=30, roundTripTimeNanos=23130000}]}
```

It basically tells us that Spring Boot configured a `MongoClient` that connects to a MongoDB Server at *localhost:27017*. Note that beans are **unique by name**, which defaults to the **type** of the bean. Therefore, there can only be one bean of type `MongoClient` in a Spring application unless you specify another name for it.

Exercise 7: Using MongoDB @Document to create Database Entities

Now that we know how to set up a connection to a SQL database, we want to conveniently interact with it directly from our Spring project. Therefore, we need to work with a few standard technologies allowing to write Java code which is then mapped to respective NoSQL code that NoSQL databases will understand as they do not know how to interpret Java code natively. Ultimately, this will allow us to define a `Project` Java class which will then correspond to a project *Collection* in MongoDB holding the different project *Documents* persisted by users.

Spring Data Repository[10] uses the configuration and code of JPA implementations (and used by default in Spring) that lets you define and query MongoDB Documents in Java code, which is then mapped to the respective MongoDB operation (e.g., `insertOne`, `insertMany`, `find`, `update`, `remove`). The Spring Framework provides extensive support for working with MongoDB databases, from direct access using MongoTemplate to complete "object document mapping" (ODM). Spring Data provides an additional level of functionality: creating Repository implementations directly from interfaces and using conventions to generate NoSQL queries from your method names. Read more on Spring Data MongoDB in the Spring Boot [11].

- Create a new `Project.java` class in `src/main/java/edu/ase/project/model`. For simplicity, our *Project* document contains only the `id` and *name*. By default, the name of the collection is the lowercase of the class name, i.e. project in this case. To change this, specify your collection name with `@Document(collection = "your_collection_name")`:

```java
package edu.tum.ase.project.model;

import com.mongodb.lang.NonNull;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "projects")
public class Project {

    @Id
    private String id;

    @Indexed(unique = true)
    @NonNull
    private String name;

    protected Project() {}

    public Project(String name) {
        this.name = name;
    }

     // getters and setters

    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }
```

[10] Spring Data Repository: https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#repositories
[11] Spring Boot with Spring Data MongoDB: https://docs.spring.io/spring-data/mongodb/docs/3.2.5/reference/html/#reference

```
33
34              public void setName(String name) {
35                  this.name = name;
36              }
37
38          }
39
40      }
```

Notice how the JPA annotations are used to create a mapping to a database collection named `project` with columns `id` and `name`. You may wonder where is the `project` collection, the answer will be revealed very soon.

Exercise 8: Create a Spring Data Repository to Query the Document

Next, create a Spring Data repository bean `ProjectRepository.java` in `src/main/java/edu/ase/project/repository` to query the `Project` Document:

```
1         package edu.tum.ase.project.repository;
2
3         import edu.tum.ase.project.model.Project;
4         import org.springframework.data.mongodb.repository.MongoRepository;
5         import org.springframework.data.mongodb.repository.Query;
6
7         import java.util.List;
8
9         // The MongoRepository is typed to the Document, and the type of the Document's ID
10        public interface ProjectRepository extends MongoRepository<Project, String> {
11            // TODO: Write a function to find the project by Name
12        }
```

The actual implementation of the `findByName(...)` method is generated by Spring Data MongoDB, which allows writing complex MongoDB queries with pure Java code. Furthermore, `MongoRepository` already provides a set of methods for querying or deleting documents using various operations, including Pagination and Sorting abilities.[12].

Exercise 9: Create a Spring Service Bean to Execute Database Operations

- Next, create a simple Spring service bean `ProjectService.java` in `src/main/java/edu/ase/project/service` to call the queries defined in `ProjectRepository`:

```
1     package edu.tum.ase.project.service;
2
3     import edu.tum.ase.project.model.Project;
4     import edu.tum.ase.project.repository.ProjectRepository;
5     import org.springframework.beans.factory.annotation.Autowired;
6     import org.springframework.stereotype.Service;
7
8     import java.util.List;
9
10    @Service
11    public class ProjectService {
12      @Autowired
13      private ProjectRepository projectRepository;
14
15      public Project createProject(Project project) {
16        // TODO: implement
17        return null;
18      }
19
20      public Project findByName(String name) {
21        // TODO: implement
22        return null;
23      }
24
25      public List<Project> getAllProjects() {
26        // TODO: implement
27        return new ArrayList<Project>();
28      }
29    }
```

[12] MongoRepository API: https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#mongodb.repositories.queries

Implement the missing parts using respective (default) methods of `ProjectRepository` or rather from the parent interfaces `CrudRepository`[13].

- After finishing the implementation, Add the `@EnableMongoRepositories(basePackageClasses = ProjectRepository.class)` to inform Spring the `ProjectApplication` that `MongoRepository` is used and the targeted repository is `ProjectRepository.class`. Next, test your service and repository in the `run(...)` method again by `@Autowired` the service class and call the respective methods.

```java
package edu.tum.ase.project;

import com.mongodb.client.MongoClient;
import edu.tum.ase.project.model.Project;
import edu.tum.ase.project.repository.ProjectRepository;
import edu.tum.ase.project.service.ProjectService;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;

import java.util.List;

@SpringBootApplication
@EnableMongoRepositories(basePackageClasses = {ProjectRepository.class})
public class ProjectApplication implements CommandLineRunner {

        @Autowired
        MongoClient mongoClient;

        @Autowired
        ProjectService projectService;

        private static final Logger log = LoggerFactory.getLogger(ProjectApplication.class);

        public static void main(String[] args) {
                SpringApplication.run(ProjectApplication.class, args);
        }

        @Override
        public void run(String... args) throws Exception {
                log.info("MongoClient = " + mongoClient.getClusterDescription());

                String projectName = "ASE Delivery";

                Project project = projectService.createProject(new Project(projectName));

                log.info(String.format("Project %s is created with id %s",
                                project.getName(),
                                project.getId()));

                Project aseDeliveryProject = projectService.findByName(projectName);

                log.info(String.format("Found Project %s with id %s",
                                project.getName(),
                                project.getId()));

                List<Project> projectList = projectService.getAllProjects();
                log.info("Number of Project in Database is " + projectList.size());
        }
}
```

- The logs should show something like the following:

```
2021-09-22 20:10:17.720  INFO 17216 --- edu.tum.ase.project.ProjectApplication   :
    Project ASE Delivery is created with id 614b7189b272651a7a13ef34
2021-09-22 20:10:17.756  INFO 17216 --- edu.tum.ase.project.ProjectApplication   : Found
    Project ASE Delivery with id 614b7189b272651a7a13ef34
2021-09-22 20:10:17.759  INFO 17216 --- edu.tum.ase.project.ProjectApplication   : Number
     of Project in Database is 1
```

[13] CrudRepository API: https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/#repositories.core-concepts

- You will see in your `aseProject` database that the *projects* collection is created automatically along with the project document. This magically means you can conveniently define the collection and document schema from the Java code without interacting with the MongoDB shell beforehand.

Exercise 10: Create a Spring Controller to Receive REST Request and Provide Services

Finally, we do not want to run our service from the command line, so let us create a REST controller to serve client requests. Do not forget to comment the codes for testing the Project Service ;)

Create a simple Spring Controller bean `ProjectController.java` in `src/main/java/edu/ase/project/controller` to receive REST requests from the client. `@Autowired` the `ProjectService` to call the available services.

We will offer the services for creating a project, get all projects in the database, and find a project with a given name. Use the annotation `@RestController` for utilizing the REST configuration methods provided by Spring. To specify the REST endpoint for our project.

```
1
2  package edu.tum.ase.project.controller;
3
4  import edu.tum.ase.project.model.Project;
5  import edu.tum.ase.project.service.ProjectService;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.web.bind.annotation.GetMapping;
8  import org.springframework.web.bind.annotation.RequestMapping;
9  import org.springframework.web.bind.annotation.RestController;
10
11 import java.util.List;
12
13 @RestController
14 @RequestMapping("/project")
15 public class ProjectController {
16
17     @Autowired
18     ProjectService projectService;
19
20     @GetMapping("")
21     public List<Project> getAllProject() {
22         // TODO: Implement
23         return null;
24     }
25
26     // TODO: Implement a POST Endpoint to create a project with a given name
27
28     // TODO: Implement an Endpoint to find a project with a given name
29
30 }
```

- The annotator `@RequestMapping` helps you shape the endpoint address of your controller. In this case, we specify **"/project"** to express that our project service base path is *http://localhost:8080/project* (By default, Spring Boot applications run on port 8080). To define an endpoint with a given HTTP method for a particular service, we use the annotator `@<HttpMethod>Mapping(<endpoint/nodes>)`. For example, to create a GET endpoint for the get all projects service, we use @GetMapping(""). Therefore, when the client sends a GET request to *http://localhost:8080/project*, the ProjectController will return the list of all projects that the ProjectService found.
- To send REST requests to our Project application, we will use Postman https://www.postman.com/. Create a GET request to the URL *localhost:8080/project*. You should see the response body with a similar content:

```
1 [
2     {
3         "id": "614b7189b272651a7a13ef34",
4         "name": "ASE Delivery"
5     }
6 ]
```

Let us practice further by creating two other endpoint handlers in our `ProjectController`:

- Create a **GET** endpoint for finding a project from a given name. Depending on how you want to extract the name from the request parameter, path or body, you will use `@RequestParam`, `@PathVariable`[14] or `@RequestBody`[15] annotator respectively.
- Add a **POST** endpoint to create a project from a given name. The `@RequestBody` annotator is helpful to accomplish this task.

[14] @PathVariable and @RequestParam: https://www.baeldung.com/spring-requestparam-vs-pathvariable
[15] @RequestBody: https://www.baeldung.com/spring-request-response-body#@requestbody