# What is the problem?

What we are building for the group project is the start of a much bigger system/framework. After our semester is done, another group will pick up where we left off to continue working. To make it easier for the next group to continue working we need to make sure the project is using modern standards, has clean code etc.

# What is the goal?

To have a backend that is easy for future teams to pick up and work with again.

# How did we solve the problem?

First, we looked at how the Spring boot framework works and what common practices are used with it.

## Keep responsibilities of each layer simple

The app is currently divided into 3 layers: controllers, services and repositories. This is a very common design pattern for Java Spring boot applications for a number of reasons. Because of this, it's expected that the next people that start working on this probably already understand how everything works. If they don't it should not be difficult to find resources online that explain it to them.

Using a common design pattern is one thing, actually applying it is the next step. To properly apply this pattern, we defined the responsibilities of each layer:

* **Controllers**: Responsible for managing the REST interface and communicating data with the business layer.
* **Services**: Responsible for handling the business logic.
* **Repositories**: Responsible for directly handling database logic, like retrieving or updating entries.

By adhering to these responsibilities and not doing anything crazy, it will be easier for future teams to pick up the work.

## Modern standards (Dependency injection/inversion, DTO's)

## Proper naming conventions

## Use Lombok

Java is a very boilerplate-heavy language. You are constantly writing the same thing over and over again for different use cases. For example, if you make a class and want to have public setters and getters, you have to write these methods for every property separately. You also have to manually write the multiple different constructors for every use case. This makes the code way longer and harder to read.

To make the code more readable, and less difficult to work with a lot of people use the Lombok package. Lombok provides a lot of decorators that automatically adds these boilerplate-heavy functions. Instead of writing different getter methods for every single property and having to remember to add one for every new property, you can simply put @Getter at the top of the class to automatically add it.

Using the Lombok library in the right way makes the code way less bloated and easier to work with. Lombok is also a really popular library so a future team should not have much trouble understanding what's going on. Even if they do not know the library, the syntax is easy to understand and there is plenty of documentation available.

For more info about Lombok, see <https://projectlombok.org/>.

# What is the result?

## Simple responsibilities

This is an example of one of the SurveyController's 'get' methods. Logic is kept to a minimum to only check for irregularities so that it can return the appropriate response code. Other than that, it just transfers the data to a method in the business logic. The responsibilities are kept to just handling the requests and nothing else.

**Controller:**

@GetMapping("/{id}")  
public ResponseEntity getSurvey(@PathVariable(value = "id") String id) {  
 if(id == null) return ResponseEntity.*badRequest*().build();  
  
 Survey survey = formsService.getForm(id);  
  
 if(survey == null) return ResponseEntity.*notFound*().build();  
  
 return ResponseEntity.*ok*(toSurveyDTO(survey));  
}

**Services:**

The services are there just to handle the business layer, which in most cases just means calling one of the repository methods. For example, the controller method above calls the following method:

public Survey getSurvey(String id) {  
 return surveyRepository.getSurvey(id);  
}

In some cases, it has to call multiple database entries to satisfy the business logic:

public Survey changeSurveyState(String id, SurveyState newState) {  
 Survey survey = surveyRepository.getSurvey(id);  
 if (survey == null) return null;  
  
 survey.setState(newState);  
  
 return surveyRepository.updateSurvey(survey);  
}

**Repositories:**

Here there is still some logic but it's actually related to the database query, as it should only return the data if it's valid and not empty. Other than that, the layer only does database queries.

public Survey getSurvey(String id) {  
 try {  
 Optional<Survey> dbResponse = database.findById(id);  
 return dbResponse.isEmpty() ? null : dbResponse.get();  
 } catch (Exception ex) {  
 System.*out*.println(ex);  
 return null;  
 }  
}

## Lombok

This is how the Survey class used to look like. Despite only having 3 properties, the class file is massive and difficult to maintain. If you want to add a property, you have to add it to the list, create a getter and setter and add it to every constructor where it's necessary. With all these added steps it's easy to forget something and to adhere the code style. It also just makes the file difficult to read.

public class Survey {  
  
 private String id;  
 private String title;  
 private String description;  
  
 public Survey(String title, String description) {  
 this.title = title;  
 this.description = description;  
 }  
  
 public Survey(String id, String title, String description) {  
 this.id = id;  
 this.title = title;  
 this.description = description;  
 }  
  
 public Survey() {}  
  
 public Long getId() {  
 return this.id;  
 }  
 public void setId(Long id) {  
 this.id = id;  
 }  
 public String getTitle() {  
 return this.title;  
 }  
 public void setTitle(String title) {  
 this.title = title;  
 }  
 public String getDescription() {  
 return this.description;  
 }  
 public void setDescription(String description) {  
 this.description = description;  
 }  
}

Using Lombok, we easily make this file way, way smaller and easier to maintain. If you want to add a property you only have to add it to the list of private properties and maybe into the custom constructor. For future developers, this is way easier to maintain and work with.

@Getter @Setter @NoArgsConstructor @AllArgsConstructor  
public class Survey {  
 private String id;  
 private String title;  
 private String description;  
  
 public Survey(String title, String description) {  
 this.title = title;  
 this.description = description;  
 }  
}

# What is the next step?

# Conclusion

# Sources

* <https://www.springcloud.io/post/2022-08/springboot-best-practices/#gsc.tab=0>
* <https://medium.com/@raviyasas/spring-boot-best-practices-for-developers-3f3bdffa0090>
* <https://projectlombok.org/>