TO-DO LIST WEB APPLICATION PROJECT (TDL)

DANIEL AHUCHOGU

INTRODUCTION

Aims:

- To build a To-Do List web application with a fully working CRUD functionality.
- Use technologies learnt from the duration of the course, to build the application.
- A working frontend and a working backend application. Using SpringBoot alongside with a java eclipse IDE for the backend side.

Objectives:

- Build the project/application by using maven. To get a working war file within the command line.
- Test the functionality of the code by conducting JUnit, Mockito and integration testing.
- To achieve a industry test coverage of 80% or higher.
- Meet the Minimum Value Product (MVP)

CONCEPT OF THE APPLICATION

- The main aspects of the project is to build a To-do List
- Researched what a To-do list is actually does and previous examples.
- Using SpringBoot to design a To-do list. For example to create a list of tasks needed to complete on a weekend.
- This is mapped using an ERD to see the relationship between the entities required.
- Simplify the specification to reach the requirements of the MVP to reach the specification goals.
- Simple user stories/user requirements that implements the aspect of CRUD.

USER STORIES/REQUIREMENTS

1

As a user I want to use the website, So that I can CREATE a To-Do list when I am on the website.

2

As a user I want to use the website, So that I can UPDATE a To-Do list with specific information related to the user when I am on the website.

3

As a user I want to use the website, So that I can DELETE a completed To-Do list when I am on the website.

4

As a user I want to use the website, So that I can READ a To-Do list when I am on the website.

5

As a user I want to use the website, So that I can READ a To-Do list with an Id when I am on the website.

ENTITY RELATIONSHIP DIAGRAMS

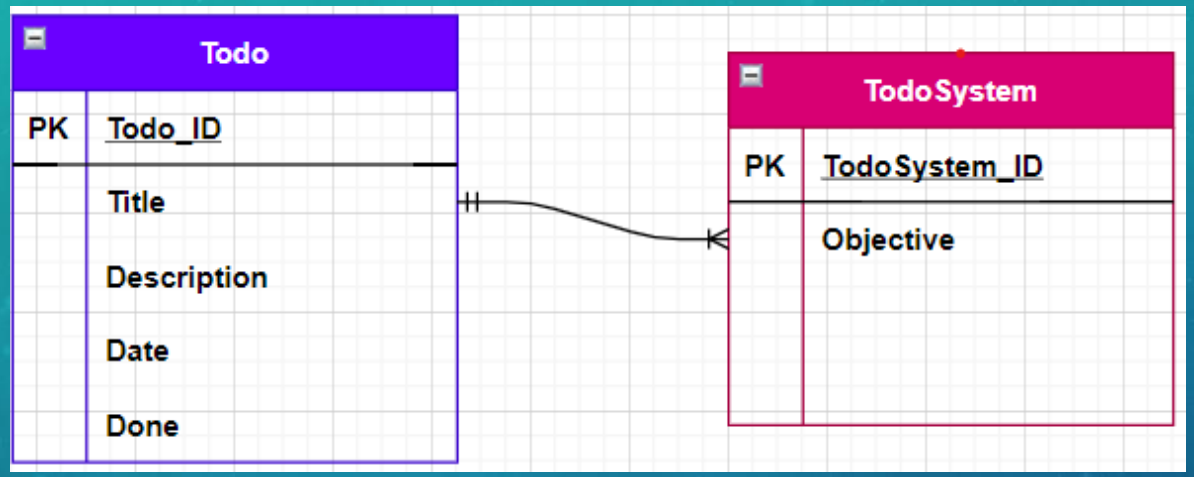


Fig 1: ERD diagram

- **Entity relationship diagram (ERD) shows the relationship of entities stored in a database**
- **We use this model to plan/design the database. To prevent many-to-many relationships we have intermediate table to handle this.**
- **Above the relationship shows a 1-to-many and many-to-many relationship**

SPRINT PLAN

- What needed to be included? What did you hope to achieve?
- Fully working CRUD functionality
- Test coverage of the src/main/java folder, above or equal to 80%
- Acceptance and Integration testing of the project (by reference the v-model).
- A version control system, with reference to the feature branch model (i.e. not working on master/main)
- A fully working war file which can be deployed from the command line interface.
- A detail project management board with before and after information with completed tasks.
- Risk assessment, ERD diagram and UML diagram.

CONSULTANT JOURNEY

The main technologies learnt and used for this project were:

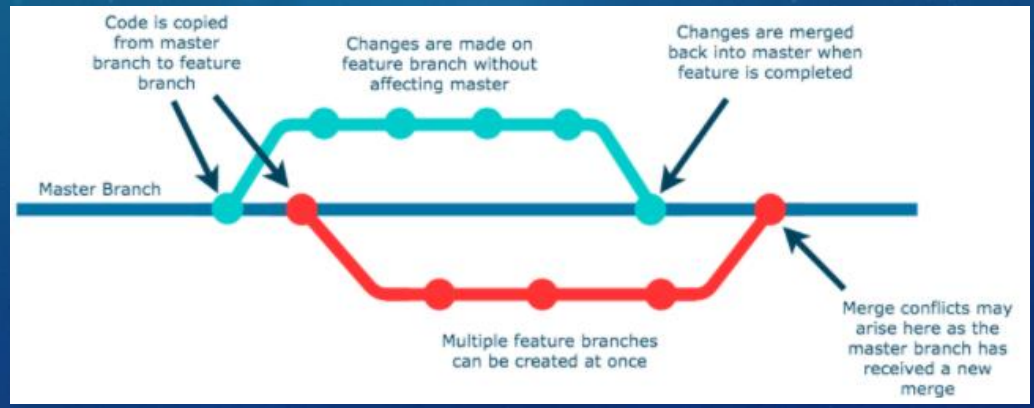
- Git, GitHub, MySQL workbench, Jira
- Java, Eclipse, Postman, Visual Studio Code, SpringBoot

Technologies:

- Git is a version control system, where it can track changes throughout the project with the Involvement of GitHub.
- SQL/GCP with SQL: Is a database management system, to manage data within tables.
- Jira: A project management system, to have the story points for all the user stories and requirements.
- Java with Eclipse IDE: A back end build tool software. Main purpose is an object orientated programming language that is class based.
- SpringBoot: Alongside Java and Eclipse creating API and creating DB relationships.
- Postman: A System to test my API's (To check if they work)
- Visual Studio Code: IDE to design the frontend i.e. JavaScript and The HTML side.

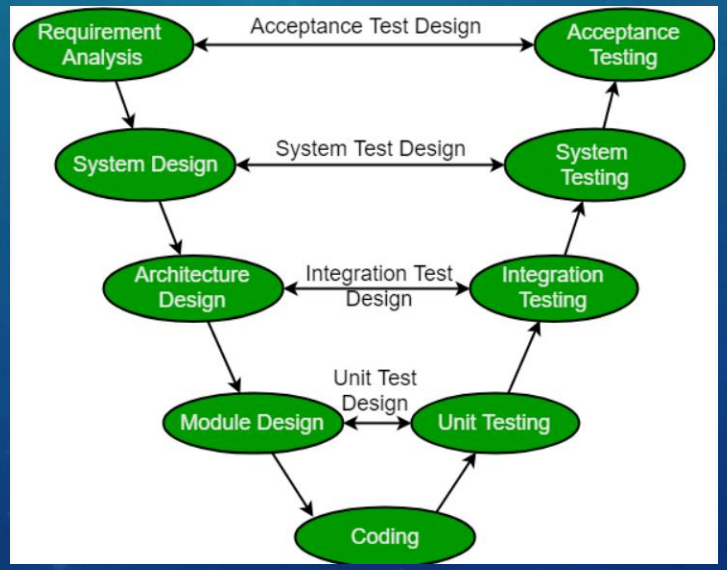
CLOUD INTEGRATION

- How did you approach version control?
- With version control, I approached it as simplest as possible to prevent errors with my code/working.
- Focused working off my DEV branch, to make different feature branches that had different aspects of my work.
- Essentially I didn't want work on my main/master branches. I.e. the need for multiple branches.
- Want to minimised has much errors and merged conflicts as possible.
- Due to different features are combined, or integrated, frequently occasionally to expose external coding issues.
- I had multiple branches, because during development you might stop working on a piece of work and restart it on a different branch and works better..



PROJECT TESTING

- For this project it follows the SDLC V model.
- In terms of testing, its always essential to test the quality and functionality of the code. i.e. less chance of the code breaking.
- JUnit testing, test the small chunks of the code and see if the code passes or fails.
- JUnit testing finds bugs in the code, which makes it crucial, in terms of development.
- Integration testing, test the individual units are tested together to check if all the code units interact with each other. SpringBoot allows Integration System Testing.
- The coverage with src/main/java is at 95% with out integration test being involved in terms of the unit code coverage



DEMONSTRATION

- With maven, the command mvn clean package. The project builds using the dependences on the pom.xml file.
- This leads to a war file similar to a jar file.
- SpringBoot used to designed the API of the website
- The controller of my SpringBoot has all the relative information that gives the websites the relative functions

```
@PostMapping("/create")
public ResponseEntity<TodoDTO> createTodo(@RequestBody TodoDomain todo) {
    return new ResponseEntity<TodoDTO>(this.service.createTodo(todo), HttpStatus.CREATED);
}

// PUT = UPDATE
@PutMapping("/update/{id}")
public Todo updateTodo(@PathVariable("id") Long id, @RequestBody Todo todo) {
    this.todoList.remove(id.intValue());
    this.todoList.add(id.intValue(), todo);
    return this.todoList.get(id.intValue());
}

@PutMapping("/update/{id}")
public ResponseEntity <TodoDTO> updateTodo(@PathVariable("id") Long id, @RequestBody TodoDomain todo) {
    return new ResponseEntity<TodoDTO>(this.service.updateTodo(id, todo), HttpStatus.ACCEPTED);
}

// DELETE
@DeleteMapping("/delete/{id}")
public Todo deleteTodo(@PathVariable("id") Long id) {
    return todoList.remove(id.intValue());
}

@DeleteMapping("/delete/{id}")
public ResponseEntity<Object> deleteTodo(@PathVariable("id") Long id) {
    return this.service.delete(id) ?
            new ResponseEntity<>(HttpStatus.NO_CONTENT):
            new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
}
```

```
@GetMapping("/readAll")
public ResponseEntity<List<TodoSystemDTO>> readAll() {
    return new ResponseEntity<List<TodoSystemDTO>>(this.service.readAll(), HttpStatus.OK);
}


@GetMapping("/read/{id}")
public ResponseEntity<TodoSystemDTO> readTodo(@PathVariable("id") Long id) {
    return new ResponseEntity<TodoSystemDTO>(this.service.readTodo(id), HttpStatus.ACCEPTED);
}

// POST= CREATE


@PostMapping("/create")
public ResponseEntity<TodoSystemDTO> createTodo(@RequestBody TodoSystemDomain model) {
    return new ResponseEntity<TodoSystemDTO>(this.service.createTodo(model), HttpStatus.CREATED);
}

// PUT = UPDATE

@PutMapping("/update/{id}")
public ResponseEntity <TodoSystemDTO> updateTodo(@PathVariable("id") Long id, @RequestBody TodoSystemDomain model) {
    return new ResponseEntity<TodoSystemDTO>(this.service.updateTodo(id, model), HttpStatus.ACCEPTED);
}

// DELETE

@DeleteMapping("/delete/{id}")
public ResponseEntity<Object> delete(@PathVariable("id") Long id) {
    return this.service.delete(id) ?
            new ResponseEntity<>(HttpStatus.NO_CONTENT):
            new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
```

SPRINT REVIEW

COMPLETION:

- Working CRUD functionalities that links with my user stories and requirements.
- Connects to a local database.
- Unit and Integration testing that is aims for a coverage around 80%.
- Git repository with the feature branch model, with a working war file.
- A basic frontend design to fetch API from the backend.

WHAT GOT LEFT BEHIND/COMPLICATIONS:

- Acceptance testing with regards to the users (essentially testing the frontend) with use of Selenium.
- The ability to call the history for some of Todo's.
- SonarQube testing (static analysis)
- CSS
- Testing if the database can be opened on other Systems.

SPRINT RETROSPECTIVE

What went well:

- Implementing CRUD architecture
- Unit testing and aspects of integration testing.
- Basic HTML design.

What could be improved:

- Some aspects of the frontend i.e. read, delete and update.
- The design of the frontend, the JavaScript, HTML and CSS.
- Spending more time on the integration testing for the controller classes.
- Time management, attempt the acceptance-user test and static analysis.

CONCLUSION/SUMMARY

- **To reflect on the project, as a whole the project went well to a certain extent.**
- **There were certain challenges and risks that occurred.**
- **But with an agile mindset I was able to adapt to the situation and I was able to work effectively.**
- **Able to achieve most of what the specification required/ in the MVP .**

Final Thoughts:

- This was a tough project for someone who has a no prior coding experience.
- This is a learn step for me to gain new skill/developed new skills that I have required.

ANY QUESTIONS?