Software Quality
Ola 1
End-to-End Testing Strategy and Implementation

By Kristofer, Mads, Michael and Søren

Test Strategy Design:

Scope and Overview:

For the first subject of our strategy we must create an overview of our project, as well as any information on who should be using this strategy. We should include information as to who will evaluate and approve strategy and lastly we should define the testing activities and phases that will be performed.

The project is the development of a To-Do board application containing different lists for organizations, an add button for creating new to-do task, and for the individual task there are a delete button for removing the task, a complete button for changing the status of the task to completed, a text field which can be edited by the users. All of which will update a MongoDB database containing the tasks. The project will be split into three layers: Frontend, backend and database. While the MongoDB database will be cloud based and be assessesable via. the internet, the frontend and backend will both be deployed locally on the same machine.
Our testing will be done as group work and the review and improvement should be done by someone from, for example: System Administration Team, Project Management Team, Development Team, Business Team, and as such we, the development team, will be reviewing and approving it ourselves.
Lastly since we only have a few days for the work assignment, our timeline for our test will follow the development of our application. We first create the test on paper then we develop them as needed as the application evolves. Meaning we create our unit tests on paper then using mock functions and then fully develop the functions after.

Testing Methodology:

For the second subject of our strategy we specify the degrees of testing, testing procedures, roles and duties of our team members. We should define the testing process, testing levels, roles and the duties of each of our team members. We should also describe why each of the test types defined in the test plan should be performed, as well as any details such as when to begin, the test owner, the responsibilities, testing approach, and the details of automation strategy and any tools if any is used.

For our project we take point in using the tests described in the assignment work document. Within the document we can see that there are three different types of tests, that being **Unit testing**, **Integration testing** and lastly **Specification-based testing**. It is also in that order we will be making the tests as it would then follow the development of the application.

For **Unit testing** we will be focusing on two areas. Testing of api calls and testing of functions. All within the backend. They should firstly be made on paper based on the user

stories created based on the assignment document with a focus on the functionality of the final application.Other than to help detect faults in our systems, they also help create the system where in the tests is created before the final function and that the final function is made based on the test mock function. The team member who is working on setting up the function is also the one responsible for handling the test related to said function. The test should return successful before continuing to work on the final function and continue to be used when running system wide testing.

For **Integration testing** we will create tests focused on the interaction between our backend and our MongoDB database. Unlike unit tests we will not be using mock functions or a mock database, instead we would normally use a development database that contains mock data, but due to the scale of the project we will just be using our production database for time efficiency. Again will the ownership and responsibility of the test belong to the team member who is developing the function. Since we aren't using a development database, will we manually trigger the tests when the database isn't actively being used by an active instance of our backend.

Lastly we have **Specification-based testing** which we will use to test the final state of our application. This test will take the form of the user of our application, meaning it will go through the frontend as if it were an actual person, and it will perform a specified set of instructions meant to mimic an actual user's journey through our application as one would use it. Unlike the other tests, this one doesn't connect directly to a specific function and as such not one single team member will be the owner for this test. Instead the entire team will be responsible for it. The test will make use of an automation tool that can emulate the application and interact with the frontend.

Testing Environment Specifications:

The specification of the testing environment we have to define the test data requirements, with clear instructions on how to prejudice the test data. Overall this section should contain information on the number of environments and the required setup, the strategies for backup and restoration as well. Some of the data we will define for our test strategy is the number of environments and their individual required configuration, as well as the number of supported users within each individual environment, including the user's access roles, software and hardware requirements including their operating system, ram, free disc and the number of systems. We also need to define the data needed for each individual test with specific instructions on how to create the test data, using either generated data or using production data by masking fields for privacy, and define a backup and restoration strategy as well. Due to unhandled circumstances in the code, the test environment database may encounter issues, so the backup and restoration method should state who will take backups when backups should be taken, what should be included in the backups, when the database should be restored, who will restore it, and what data masking procedures should be implemented if the database is restored.

Since we are using GitHub version control to store our project it also makes sense to make use of their GitLab for continuous integration, where we can set up a hosted testing environment that automatically we run our test, and deploy it if we later on find the need

and/or want. When setting up the virtual environment for GitHub-hosted runners we have five different options.

Standard GitHub-hosted runners for public repositories
• Linux with 4 processors, 16 GB ram, 14 GB ssd storage and the labels: ubuntu-latest, ubuntu-24.04 [Beta], ubuntu-22.04, ubuntu-20.04. • Windows with 4 processors, 16 GB ram, 14 GB ssd storage and the labels: windows-latest, windows-2022, windows-2019. • MacOS with 4 processor, 14 GB ram, 14 GB ssd storage with the label: macos-13. • MacOS with 3 processor, 14 GB ram, 14 GB ssd storage with the label: macos-12. • MacOS with 3 (M1) processor, 7 GB ram, 14 GB ssd storage with the labels: macos-latest or macos-14

Considering the test run within their own environments within the tools we will use, we have chosen to use the Linux option as we have learned over the course of our education that Linux is the preferred operating system for hosting operations. Within the test environment we will be testing all of our tests, the unit tests, the integration tests and the specification-based test. For each test type we would create a task that will run on the virtual environment. Within the task would be console commands that will run our testing tools. The tests themselves will contain the data needed for running themselves, and considering that, will we set up a development database where any data stored within will be erased before the tests run and the tests themselves will also clean up after themselves when completed.

Testing tools:

When it comes to the tools we may use for our testing, we must define what tools for test management and automation that we will use for the tests, describe the test approach and the tools needed for performance, load and security testing, as well as mention whether the product is open-source or commercial, and the number of individuals it can accommodate.

For the tools we intend to use for our test, we will be using different tools that we have chosen from experience. The two tools have been used by multiple members of our team previously in work assignments for our previous education. The two tools we will be using are Jest and Cypress.

Jest will be used for our unit test and integration tests. It’s a testing framework which focuses on simplicity and works with typescript, which we use to make our application. It’s capable of generating code coverage by the addition of flags and makes use of a custom resolver for imports in our tests which makes it simple to mock up.

Cypress will be used for our single specification-based test. This testing framework makes it possible to simulate the users experience with simulated input for our final running application version. It allows us to create a set of actions that follows the user stories we create and the test is considered successful if all actions can be completed and if the simulated application successfully updates to reflect the actions.

Release Control:

The release control is used to make sure that the test execution and release management strategies are established in a systematic way. To do that we need to specify different software versions in the test and UAT environments that can occur from unplanned release cycles. And that all adjustments in that release will be tested using the release management strategy, which includes a proper version history. And finally set up a build management process that answers any questions like where the new build should be made available, where it should be deployed, when to receive the new build, where to acquire the production build, who will give the go signal for the production release.

With the use of GitHub and GitLab we have previously determined how we would run automatic testing, and we can build on that to complete our CI/CD. Building on the task we use to test, we can add a task that will update and deploy the newest version of our application based on a specific branch of our version control. By creating a branch, that we for example could call “Deployed”, we could manual update that branch using the “main” branch where after GitHub/GitLab would automatically start our testing and upon completion of the tests and a 100% successful result then upload the new build, which would also be build within the virtual environment, and deploy it on a remote online hosting server like that of Digital Ocean. We could also have it store the build on our GitHub repository so that we can manually download and run the application locally.

Risk Analysis:

In this section we need to go over the potential hazards that are associated with our project that can become an issue during the test execution. We need to make a list of the potential dangers, provide a detailed plan to manage the risks and a backup in case the potential dangers become reality.

We will create a list of potential risks that we have considered which will follow with a management strategy about minimizing the chance of it happening and finally a backup on how to handle the risk should it occur.

	Risk Description	Risk Management:	Backup Plan
1	The remote MongoDB database crashes due to internal error.	We can’t reduce the risk of the database shutting down due to an internal fault.	Manually restart the database when available.

Test Plan:

Create a test strategy that includes unit tests, integration tests, and specification-based tests and discuss how each type of test contributes to the overall quality of the software.

Before creating any test, we decided to set up some user stories made based on the requirements listed in the project pdf. We believe that the user stories will help us create better tests and also help us decide where and how each test should be made.

User stories:

1. As a user, I want to be able to add a task to my to-do list, so that I can create a list of what I need to do.

2. As a user, I want to be able to see a list of all the tasks on my to-do list, so that I can have an overview of everything I need to do.

3. As a user, I want to be able to edit the text of a to-do card, so that the right information is saved.

4. As a user, I want to be able to delete a specific task on the to-do board, so irrelevant information that I don't need to remember isn't saved.

5. As a user, I want to be able to mark a task as completed, so that I can see which tasks I have completed and which task I still need to do.

6. As a user, I want to be able to move a card between different lists, so that I can organize my to-do board.

7. As a user, I want to be able to set a deadline for a task, so that I can set a time limit on my tasks.

Unit Tests:

Firstly we want to, using unit tests, create mock functions for the different functionalities of our application. It is generally considered good practice to first create unit tests with mock functions before creating the final functions and it is therefore why we first will set up unit tests.

- Test 1 - Adding a to-do task. - Test API

Function	Successful result
AddTask creates a mock task and then attempts to create a new task containing the same text.	The mock task contains the same values as the created task.

- Test 2 - Deleting a to-do task. - Test API

Function	Successful result
DeleteTask sends along an ID of the task that is meant to be deleted, finds the task at hand and continues to delete it.	When the selected task with the id sent is deleted.

- Test 3 - Edit to-do task text. - Test API

Function	Successful result
EditTask creates a mock task and then first attempt to update an already existing task with the same id to match all values	The mock task contains the same values as the updated task.

- Test 4 - Edit to-do task is complete state. - Test Function

Function	Successful result
EditCompleteState sends an ID of the task that is desired to be edited and the current “completion state” of the task. Based on this, the state is updated to the opposite on the task with the matching ID.	If the completion state of the targeted task is set to the opposite of the current state.

- Test 5 - Edit to-do task list position (Category ID). - Test Function

Function	Successful result
EditTask creates a mock task and then first attempt to update an already existing task with the same id to match all values	The mock task contains the same values as the updated task.

- Test 6 - Edit to-do task deadline. - Test Function

Function	Successful result
EditTask creates a mock task and then first attempt to update an already existing task with the same id to match all values	The mock task contains the same values as the updated task.

Integration Tests:

For integration tests, we aim to verify the interactions and the data exchange between the different system components and/or modules in our software application. The goal of integration testing is to identify any problems or bugs in our system that might arise when different components are combined and interacting with each other.
The difference between standard unit tests and integration tests is that while unit tests remain internal for each component in the system, the integration tests revolve around the interaction of components.

- Test 1 - Moving to-do tasks to different category:

Function	Successful result

- Test 2 - Updating to-do tasks text:

Function	Successful result
EditTask creates a dummy task and then first attempt to update an already existing task with the same id to match all values	The mock task was successfully updated in the db and contains the same values as the updated task.

- Test 3 -adding task:

Function	Successful result
AddTask creates a mock task and then attempts to create a new task containing the same text.	The mock task was successfully created in the database and contains the same values as the created task.

Specification-based Tests:

With specification-based testing we take point around testing from the users point of view. We will test rather or not if each of our user stories can be accomplished.

- Test 1 - Average user usage.

Function	Successful result
The tests starts by redirecting to the home page to the website at http://localhost:3000	This step is successful when Cypress arrives at the desired homepage.
Takes a hold of the input fields with text and deadline and continues to write credentials for a new task and ends this step by clicking “Add task”.	This step is successful when Cypress manages to create a task with the information provided in the input fields.
When it’s created, the dropdown menu for categories is selected and the category “Work” is selected for the newly created task.	This step is successful when Cypress assigns the “Work” category to the new task
Cypress redirects to the “Chores” tab to see if the task appears in this section.	This step is successful when Cypress is unable to find the “Work” task in this section (The “Chores” section)
Cypress switches to the “Work” tab and checks if the task is there like it’s supposed to	This step is successful when Cypress finds the task in the “Work” section
Cypress looks for and deletes the newly created task	This step is successful when Cypress manages to delete the task that was created in the start of the test

Use of Test Doubles:

For each instance where we incorporated a test double, based on the principles discussed by Martin Fowler, we need to clearly explain why and how we used each type of test double.

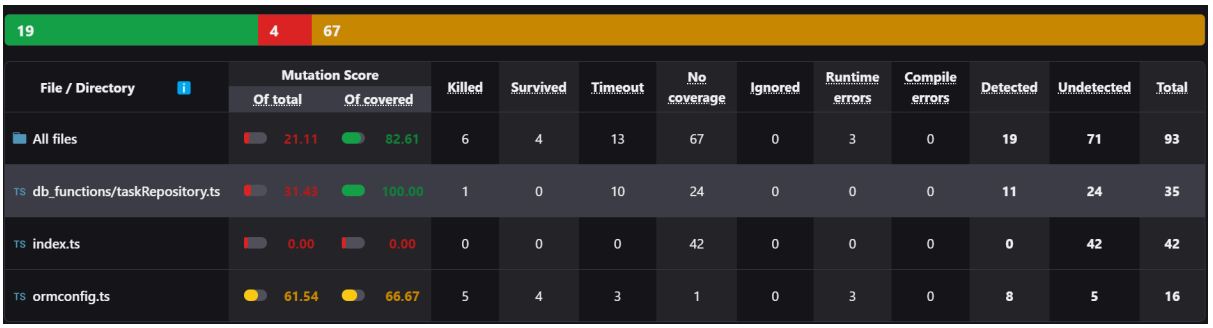
Primarily throughout our tests we make use of dummy objects, stubs and mocks. For all of our unit tests we start by creating a dummy object that is solely used in equal statements to determine rather or not the test was successful. Together with our dummy objects we make use of mock functions as they only provide answers during the test and are pre-programmed with expectations of what they will receive.

With integration testing we again make use of dummy objects for containing the values to be used with equal statements and we make use of stubs by creating local methods within the tests themselves. We also make use of mock functions in the same way as the unit tests where we use the dummy objects as values for them.

Lastly we have our specification-based test which aren’t made the same way, that being through programming, and instead is created within the tool Cypress. As it simulates the final application, it doesn't use any extra code and because of that we don’t see how any of the points made by Martin Fowler fits this test.

Analysation of validation through Mutation Testing:

Mutation testing was applied to validate the effectiveness of our tests by introducing small changes(mutations) to the code. By assessing if the tests fail or not, we can asses and simulate common mistakes or bugs that could occur during development. To validate we applied mutation testing using Stryker.



19 | 4 | 67

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
All files	21.11	82.61	6	4	13	67	0	3	0	19	71	93
db_functions/taskRepository.ts	31.43	100.00	1	0	10	24	0	0	0	11	24	35
index.ts	0.00	0.00	0	0	0	42	0	0	0	0	42	42
ormconfig.ts	61.54	66.67	5	4	3	1	0	3	0	8	5	16

By running Styker across our project we generated 93 mutations. The results showed that 19 of these mutations were detected and killed by our existing tests, meaning our tests successfully identified and failed on these mutations, while only 4 mutations survived. This suggest that some areas are not robust enough. Another concern is 67 mutations were not covered, indicating that chunks of our codebase are not covered by tests.

Reflection and Discussion:

Reflecting on Verification vs. Validation:

Many people at first glance could consider verification and validation to mean the same thing but knowing when and where to use what can help us make a better assessment of the work we face. IEEE-STD-610, which is a software engineering standards, defines them as:
Verification, “A test of a system to prove that it meets all its specified requirements at a particular stage of its development.”
Validation, “An activity that ensures that an end product stakeholder’s true needs and expectations are met.”
This can be boiled down to a difference in focus between the two terms, with verification focusing on how we got the result and validation only focusing on that we obtained the correct result. They also each have their own place in the development process where verification takes place while a product is being developed, the validation is performed upon a completed system or on the entire application and validation is also there to ensure those who paid for the application receive what they wanted. Looking through the test we have created during this assignment, we have used verification through the use of our unit tests and our integration test, with the specification test providing validation of the final product.

Reflection on Quality:

Looking back at our initial testing strategy, it lacks more clear and defined objective metrics that go with our already tests, for example measurement of mean time to restore for our database. While we also haven’t discussed code analysis tools, we still have been passively using it throughout the development of our application, as it is active by default in the form of static code analysis, which can help find unused variables, unreachable code and more.

Discussion on Test Categories:

There are a total of 10 different test categories that we see in the article not counting the first subject which isn’t a test category in itself. From those 10 categories we make use of Unit Test, Integration Test, Story test and User Journey Test. Unit test and integration test are self explanatory as we previously discussed. It is from our one specification-based test that we go into the categories of story test and user journey test. User journey tests are described by Martin Fowler as a form of Business Facing Test designed to simulate a typical user’s “journey” through the system, and it’s within this simulated environment that we enter a story test as we, within the simulation, test if all of our created user stories can be completed.