Test plan for the application: an E-commerce store

Software testing COMP.SE.200

Teemu Ruonakoski 152116224

Aleksi Kehusmaa 150323035

Table of contents

Definitions, acronyms and abbreviations

UI	user interface
E2E	end-to-end
REST	representational State Transfer
API	application Programming Interface
AI	artificial intelligence
RTL	React testing library

1. Introduktion

This document describes three end-to-end (E2E) scenarios for an e-commerce web application. The purpose of the document is to provide a plan for the software tests for the components and the functions. This document provides an idea for the testing types and utility libraries that are relevant for each scenario. The aim is to ensure that the application works seamlessly from both the customer's and the producer's perspectives, enabling customers to purchase products as easily as possible from the e-commerce web application, where the producers can sell their products as easily as possible. This document focuses on verifying the functionality, integrations, and data flow between the front end (React) [1], back end (REST API) [2] [3], and third-party services.

Contests of this document are the three E2E scenarios that are, 1. Product search and adding to cart, 2. Product management by producer, and 3. Ordering through third-party services. 1. Describes how a customer can find products using the search engine or filter the products and then adding them to the shopping cart. This scenario verifies the user interface behavior, React component logic, data flow and price updating mechanisms. 2. Covers the process where a producer adds, edits, or removes products through the interface. The focus of this is to ensure that product information is correctly updated in both the back end system and the customer view, including data validation and user interface (UI) updates. 3. Describes how the customer completes the purchase by reviewing the cart, entering payment details, and paying through the integrated third-party payment service. Tests include price calculations, payment service integration, authentication, and confirmation display. [4]

Each of these scenarios specifies the purpose for testing, step-by-step workflow of the process being tested. The tested areas and React components. Test types that are Unit-, integration- and UI-tests, also the relevant utility library functions [5] that we have selected for this assignment. Together, these scenarios form a good of a complete E2E overview of the main use cases of this e-commerce application, from searching for a product to order confirmation.

2. Contexts of testing

2.1 End-to-end scenarios

The system of e-commerce application is a web-based system that enables customers to browse, search and purchase products that the producers list there to sell. The three E2E scenarios have been selected so that they would cover as much as possible of the most important things about this e-commerce store. Each scenario includes defined steps, testable components, and associated utility library functions. [6]

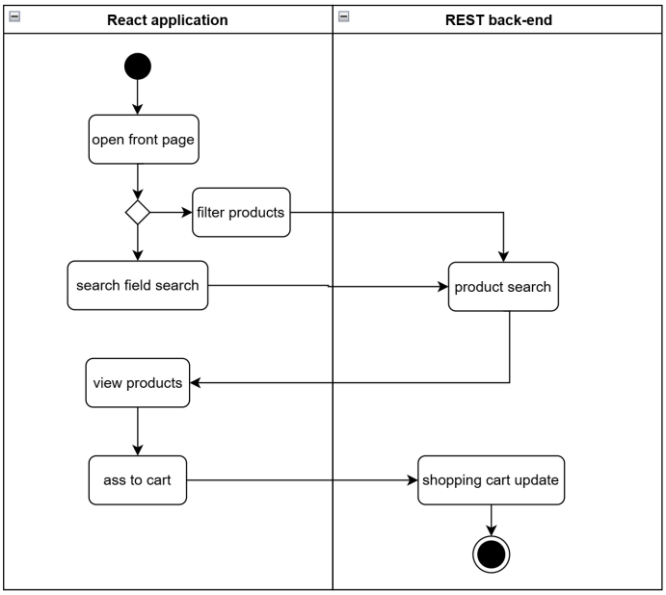

Figure 1: End-to-end scenario 1 (customer mode)

Goal, of the first scenario (Figure 1: End-to-end scenario 1 (customer mode)) is to ensure customers can find, filter, and add products to the cart, with accurate price updates. The main components of this are the search bar, product list, shopping cart, and back end API, utility library functions to these are FILTER, REDUCE, ADD, CAPITALIZE and UPPERFIRST.

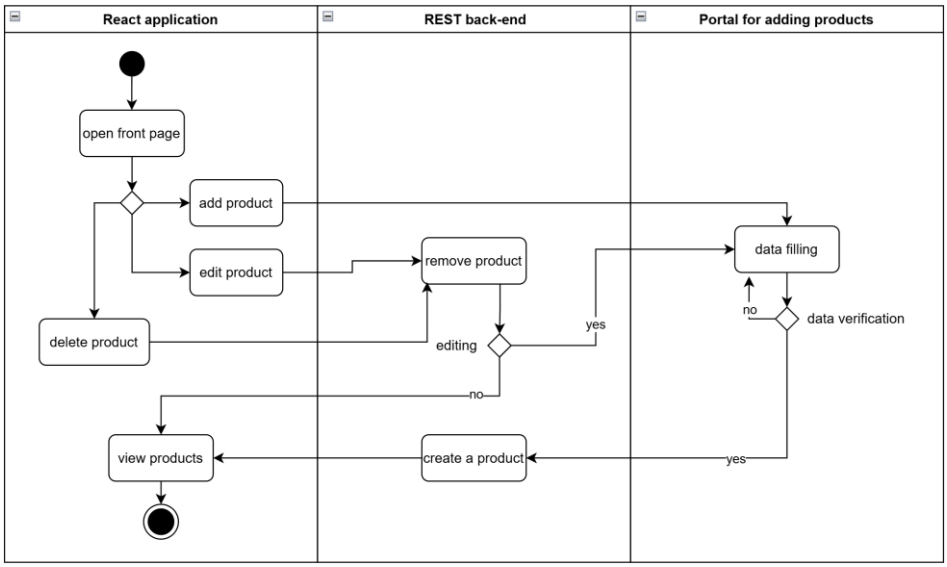

Figure 2: End-to-end scenario 2 (producer mode)

Goal of the second scenario (Figure 2: End-to-end scenario 2 (producer mode)) is to validate that producers can add, edit, or remove their products through the UI and that changes are reflected in the back end and customer view. The main components are product management view, input validation, and product data synchronization. Utility library functions to these are FILTER, CLAMP, CAPITALIZE, COMPACT, KEYS, TONUMBER.

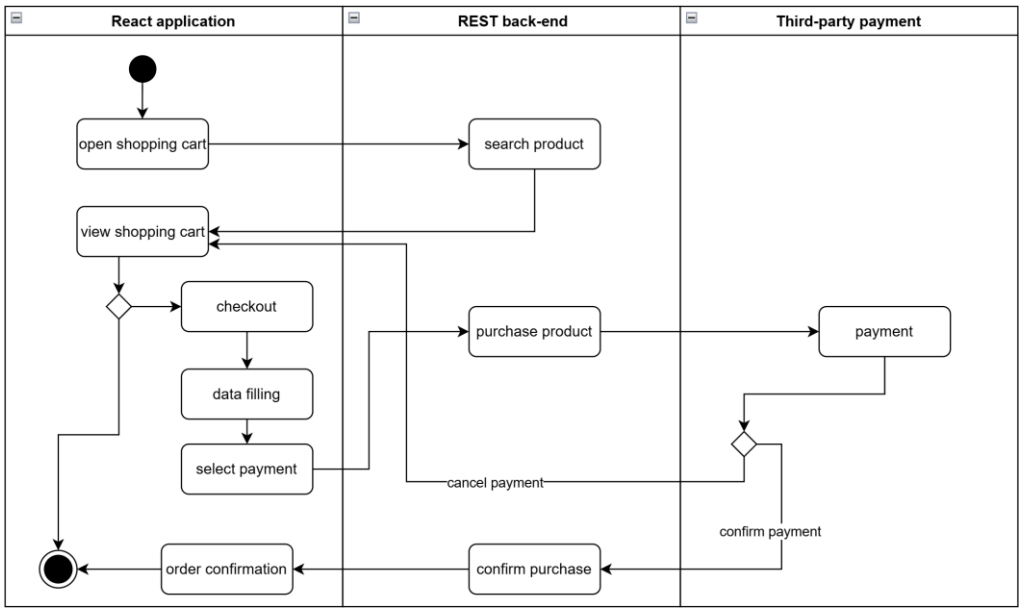

Figure 3: End-to-end scenario 3 (customer mode)

Goal of the third scenario (Figure 3: End-to-end scenario 3 (customer mode)) is to confirm that customers can order items, the total price is correctly calculated, and payment via external service is successfully integrated. The main components are cart summary, checkout form, back end API, and payment gateway interface. Utility library functions needed for this are ADD, DIVIDE, DIFFERENCE and TOSTRING.

2.2 Testing scope

In scope for this testing is the Front end, where the testing focuses on UI functionality and responsiveness of main views that consists of search, cart, checkout and producer portal. Other things that are front end related are input validation and formatting rules, and component-level state management and event handling. Back end integration related things are REST API endpoints for product search, cart management, and payment operations. Also, data flows between front end and back end. Unit testing in this case mostly focuses on selected utility library functions that directly support the main scenarios. UI tests that the overall process flows to ensure data consistency and correct user experience.

Out of scope for this testing are the internal back end logic not exposed via REST API, also the hidden internal directory in the utility library. Other things we are not considering are

performance and full security. These are important things in the e-commerce store, but when you can only choose three E2E scenarios, we consider these things less important than the things we have chosen since the three of them are very different and cover most of the important things.

In this testing we are only selecting ten utility library files, and these are in our opinion the most relevant to the defined end-to-end scenarios (see Table 3: Risk analysis for unit testing functions). Prioritization is based on risk-based testing, because we are taking the risk without testing our software with all of the utility library files. Selecting these functions that have the highest impact on our testing outcome, on these three E2E scenarios. These ten files will be the focus of unit test implementation in part 2 of this assignment.

2.3 Testing related assumptions

Assumptions in this testing are that everything we test work, but here is the breakdown on what our assumptions are: 1. The back end REST API endpoints are functional and stable for testing purposes. 2. The test environment simulates third-party payment and email services using mock on sandbox environments. 3. The front end validation logic reflects the production version of the React application.

3 Tools and reporting

3.1 Testing tools

In the second part of this assignment the actual test environment will be set up and configurated. In this first part we have selected the tools and their roles are defined in this plan. These are given in the appendix Table 4: Selected tools and their purpose.

3.2 Reporting

Reporting during and after testing ensures transparency, traceability, and continuous quality monitoring throughout the testing process. The reporting process will follow our planned approach, where the testing process will be covered by our plan. Our test process will be reported to our documents, by each test and E2E scenario. Process will include the number of tests executed, number of tests passed, code coverage metrics and current testing phase status.

3.3 Defect tracking

Defect tracking ensures that all identified defects are documented, prioritized, monitored and resolved systematically throughout the project. The process provides transparency in defect management and ensures that no critical issues remain unsolved when finishing this project. All defects will be logged and managed, which allows for clear traceability between test cases commits, and defect resolutions. Each issue will be linked to the corresponding test case, E2E scenario, and commit reference. [7] The fault life cycle is described in the Table 5: Defect lifecycle and criticality and priority levels are described in the Table 6: Criticality and priority levels. Each defect is labeled with both criticality and priority; this enables effective risk-based handling.

Defect reporting template. Every logged defect follows this structure. **Title**, the name of the issue. **Description**, explanation of the problem. **Steps to reproduce**, list of steps leading to the problem. **Expected result**, correct system behavior. **Actual result** observed system behavior. **Priority**, classification based on risk and impact. **Linked test case**, reference to the related test. **Attachments**, screenshots demonstrating the defect.

3.4 Quality gates and metrics

Quality gates define the criteria for entering and exiting each testing phase, ensuring that all test stages are completed with adequate quality and coverage before moving forward. Metrics are used to measure progress and effectiveness of the testing process.

Entry criteria, testing activities can begin when these conditions are met. 1. The front end and back end are deployed. 2. Mock or sandbox are available for third-party application. 3. Test data is prepared. 4. Jest and cypress are configured. 5. Our plan for this testing is completed.

Exit criteria, testing can be concluded when, 1. All C1 and C2 are resolved. 2. 90% Unit test coverage is achieved. 3. 80% UI coverage is achieved. 4. All three E2E-scenarios are successfully executed. 5. Test report is completed.

4 Tests

4.1 Basis of testing

In general, the goal of E-commerce store application testing is to ensure the correct functioning, usability and compatibility of the application with the interfaces. Correct functioning is ensured through unit testing and UI testing. Usability is ensured through UI-testing. Compatibility is ensured through integration testing. In addition, the number of concurrent users in an e-commerce store can increase significantly if, for example, a desired product goes on sale. Therefore, the performance of the application must be significantly higher than the average number of active users requires. This should also be considered as part of testing. In addition, ensuring cyber security would also require security tests. [8]

E-commerce store acts as a trading platform so it is related to the user's spending and purchasing of products. For this reason, proper functionality is particularly important, especially when it comes to pricing and shopping cart content. All financial irregularities require precise investigation, so their number should be minimized. The same is also related to compatibility of the application with the interfaces. The products for sale to the customer must be available, the producer's products must be visible to the customer and the payment transactions to be made must match the orders. Usability is essential when the purpose is to serve customers in their food purchases. Ordering should not take an unreasonable amount of time and searching for products should be smooth so that the customer places the order and orders again later. The same applies to the user experience of the producer. The management of products for sale must be smooth so that producers can maintain up-to-date sales content. These requirements determine the functionalities to be tested.

4.2 Tests to be performed

The above requirements apply to all defined E2E scenarios that include all the most important functions, considering the above requirements. The customer selects the products to order based on their price, availability and other information which must match the information set by the producer. When the products to be ordered have been selected they appear in the shopping cart. Contents and total price displayed in the shopping cart affects whether the customer's order is fulfilled. The actual transfer of funds takes place in the third scenario, which must work correctly so that the different party’s loss or gain the correct amount of money. So, testing based on E2E scenarios ensures the suitability of the e-commerce store application for its intended purpose.

All application activities related to prices and money transactions primary targets for testing and errors in them must be minimized. Another important test item is matching the products

and their information displayed for sale with the actual information added by the producers, so that the producer can sell and deliver and the customer can get the products they want. Usability is the least significant test item because it does not affect the actual functionality of the application.

Due to the current testing resources and environment, testing is limited to automated unit testing and in other respects narrow. Therefore, usability- and performance testing are excluded. Usability testing must be performed manually anyway, would require the inclusion of manual testing. Based on the E2E scenarios and risk analysis (see Table 3: Risk analysis for unit testing functions), the most important functions are selected as test targets to achieve the most comprehensive unit testing possible.

4.3 Detailed test case design

This section presents a test plan for two functions to be tested. Plan contains used parameters and expected results and behaviour and purpose of the test case. This also serves as a unit testing model for other functions to be tested. In mathematical functions tests should focus on extreme and limit values, as well as values that require special treatment. Functional tests should verify operation with expected parameters, as well as error parameters or other special situation parameters. The test cases below help the tester think about what types of tests are needed for different functions.

DIVIDE:

“Divide two numbers”

param {number} dividend The first number in a division

param {number} divisor The second number in a division

returns {number} Returns the quotient

Table 1: DIVIDE function test cases

situation type	param {number}	param {number}	expected return {number}
normal	6	2	3
negative divider	6	-2	-3
negative dividend	-6	2	-3
both negative	-6	-2	3
divide by zero	-6	0	error
dividend zero	0	2	0
decimal values	0.06	0.02	3
string divider	a	2	error
string dividend	2	a	error
empty param	empty	2	error

special result	1	3	0.333...

FILTER:

“Iterates over elements of `array`, returning an array of all elements `predicate` returns truthy for. The predicate is invoked with three arguments: (value, index, array)”

param {Array} array The array to iterate over.

param {Function} predicate The function invoked per iteration.

returns {Array} Returns the new filtered array

Table 2: FILTER function test cases

situation type	param {Array}	param {Function}	expected return
normal	[{'item':'meat', 'sold':false} {'item': 'potato', 'sold': 'true'}]	({ sold }) => sold	['potato']
array empty	[]	({ sold }) => sold	[[]]
predicate empty	[{'item':'meat', 'sold':false} {'item': 'potato', 'sold': 'true'}]	({}) => ()	error
function content not in array	[{'item':'meat', 'sold':false} {'item': 'potato', 'sold': 'true'}]	({ reserved }) => reserved	[[]]

4.4 AI and testing

Artificial intelligence (AI) is a powerful tool for designing and executing test cases. The strength of AI is the rapid and extensive creation of test cases, as well as the coding of the test functions themselves. Testing can therefore be significantly expanded and made more efficient with AI. It is an effective tool, especially in unit testing. It can also help in creating integration tests, if it is given the necessary information about the functionalities. In usability testing, AI cannot help because it is not able to evaluate like a human. In general, AI works best as a support for planning and ideation, when deficiencies and excesses are more easily noticed.

In the case of this plan, the advantage of using AI would be to make the testing of the functions to be tested more efficient. It could quickly create different test cases, and the tester would only need to check that everything needed is tested. In integration tests, such as testing the functionality of the Reach UI and the REST back end system interface, it could be challenging to get AI to control the whole so that it could produce test cases independently. It would still work as a tool for creating test cases, if the tester tells what needs to be tested. In mathematical matters, such as calculating results of prices, it is possible for AI to produce incorrect calculation results. Therefore, they should be produced with an actual calculator

program. When working with AI, it is always important to note that it makes mistakes, so its outputs must always be checked.

In testing the utility library functions implemented according to this plan, it is worth utilizing AI in writing the test parameters presented in tables Table 1: DIVIDE function test cases and Table 2: FILTER function test cases. This makes the actual writing work easier and leaves time to focus on creating the tests and test cases themselves.

AI-usage in this section was creating smaller chunks from the assignment to help us easily plan step-by-step how we will write this document.

5 References

[1] “React (software),” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/React_(software). [Accessed 1 11 2025].

[2] “REST,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/REST. [Accessed 1 11 2025].

[3] “API,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/API. [Accessed 1 11 2025].

[4] P. Ammann and J. Offutt, Introduction to software testing, 2nd ed., Cambridge University Press, 2016.

[5] otula, “Utility library COMP.SE.200-2024-2025-1,” [Online]. Available: https://github.com/otula/COMP.SE.200-2024-2025-1?tab=readme-ov-file.

[6] Mikrosoft Learn, Software testing life cycle (STLC) and quality gates, Mikrosoft documentation, 2025.

[7] P. Kettunen, Ohjelmistotestauksen perusteet: Suomenkielinen perusteos testausmenetelmistä ja laadunvarmistuksesta, Talentum, 2019.

[8] Testlio, “What Is E-commerce Testing: 2025 Guide,” 15 4 2025. [Online]. Available: https://testlio.com/blog/ecommerce-testing-guide/. [Accessed 4 11 2025].

6 Appendix

Table 3: Risk analysis for unit testing functions

	1	**2**	**3**	**4**	**5**
Frequency scale: 1-5 [per application launch]	rarely	occasionally	about once	many times	continuously
Criticality scale: 1-5 [per order if fails]	cosmetic harm	complicated	requires rework	ordering not possible	financial loss
Related to E2E scenario	**Function name**	**Freq**	**Crit**	**Base value**	**Category**
[1,2]	AT	1	4	4	would
[1,2]	FILTER	4	3	12	should
[1]	REDUCE	5	2	10	should
[1 2 3]	TOFINITE	1	5	5	would
[1 2 3]	ADD	5	5	25	must
[1 2]	CAPITALIZE	4	3	12	should
[1 2]	UPPERFIRST	4	3	12	should
[2]	CLAMP	3	5	15	must
[3]	DIVIDE	3	5	15	must
[2]	COMPACT	2	4	8	would
[1 2]	KEYS	2	3	6	would
[2]	TONUMBER	3	3	9	would
[3]	DIFFERENCE	3	5	15	must
[3]	TOSTRING	3	1	3	would

Table 4: Selected tools and their purpose

Tool	Purpse and role in testing	Benefit to E2E scenarios	Usage instructions document
Jest	Main JavaScript testing framework for unit and integration testing of React components and utility functions. Provides mocking, assertions, and snapshot testing features.	Supports unit and integration testing for functions like ADD, FILTER, DIVIDE and UI state updates in all three scenarios.	Jest documentation
RTL	Tool for testing React components focusing on	Verifies UI functionality in Scenario 1 (search,	RTL documentation

	user interaction and behaviour rather than implementation details.	add to cart) and Scenario 2 (product management).	
Cypress	E2E testing framework for simulating real user workflows in the browser.	Used for verifying complete flows from product search to checkout (Scenarios 1–3), ensuring UI and back end integration works correctly.	Cypress documentation
Postman	API testing tool for verifying REST endpoints and back end responses independently.	Ensures data consistency and correct API behavior between front end and back end (Scenarios 2 and 3).	Postman documentation
c8	Code coverage analysis tool integrated with Jest.	Ensures adequate unit test coverage of utility functions (ADD, DIVIDE, FILTER, etc.) and component logic.	C8 documentation
Coveralls	Cloud-based code coverage reporting and visualization tool integrated with CI/CD.	Provides visibility of test coverage progress for all scenarios. Supports continuous quality tracking.	Coveralls documentation
GitHub actions	Continuous Integration and Deployment (CI/CD) platform that automates test execution on each commit or pull request.	Runs all Jest and Cypress tests automatically for every code change, ensuring stability across all E2E flows.	GitHub actions documentation
Diagrams.net/draw.io	Online tool for creating UML sequence and activity diagrams for scenario visualization.	Used to illustrate E2E flows (Figures 1–3) and clarify component interactions in documentation.	Diagrams.net documentation
Visual Studio Code	Development and debugging environment with integrated test runner plugins.	Provides a unified interface for running Jest and Cypress tests, debugging and viewing test results.	Visual Studio Code documentation
GitHub issues	Defect tracking and reporting platform used for managing discovered defects and their lifecycle.	Tracks test defects and progress across all E2E scenarios, ensuring traceability and prioritization.	GitHub issues documentation

Table 5: Defect lifecycle

Stage	Description
New	Defect discovered during testing and logged in GitHub Issues. Awaiting triage.
Assigned	Defect assigned to a responsible developer or tester for investigation.
In progress	Developer is working on fixing the defect.
Resolved	Fix implemented and submitted for re-testing.
Re-tested	Tester verifies if the fix resolves the defect.
Closed	Defect verified as fixed and confirmed to have no side effects.
Reopened	Defect persists or reappears after being marked as closed.

Table 6: Criticality and priority levels

Criticality	Description	Example	Priority	Description
Critical (C1)	System unusable	Payment failure, data corruption	P1	Fix immediately, block further testing
High (C2)	Major function blocked	Checkout process error	P2	Fix in next release iteration
Medium (C3)	Functionally impaired	UI misalignment	P3	Fix after all critical issues are resolved
Low (C4)	Minor defect, no functional impact	Typographical error	P4	Fix only if there's enough time