Testing Approaches

Document metadata: LO1 1.3 1.4 EVI 1

Contents:

Testing Approaches

Please refer to document `LO1 1.1 EVI 1` for a statement of requirements.

R1

- **Approach: End-to-end testing**
- **Appropriateness:**
  - Unauthorised access can be gained at different levels of the system.
  - E2E is testing a complete application flow, from start to finish, including the user interface (UI), the application programming interface (API), and the database. The goal of E2E testing is to test the application as a whole and ensure that all of its components are working together correctly in respect to authentication.
  - We should simulate different scenarios where a user tries to access the database without the proper permissions at different levels and components of the system.
- **Limitations:**
  - More or less limited to UI testing: E2E testing is typically limited to testing the user interface and the interactions with the system, it does not cover the

internal workings of the system where logic errors might lie which cannot be uncovered by UI interactions.

        - Justification: in a real-world scenario, I would expand the requirement to specify that unauthorised access should not be allowed at any level of the system (not just via UI), and use e2e-testing **in conjunction** with appropriate integration and unit tests which test the same matter (unauthorised database access). However, here I want to demonstrate my ability of implementing and applying e2e-testing, so I simplified the requirement (**restriction principle**). This simplification is valid if we anticipate the user base or potential attackers to be users who only interact via the UI (which is an important limitation to be aware of).
    - Complexity: E2E-testing can be complex and time-consuming to set up as it involves simulating how user would interact with the system in a real-world scenario.
        - Justification: this would be a problem if most of my test approaches would involve large complexity as students also have a time-cost budget. However, this is not the case and therefore it's justified to have a more complex testing approach here.
    - Limited isolation: E2E-testing provides less isolation than, for example, unit testing which can make it harder to identify and trace back the source of an error.
        - Justification: as mentioned before, in a real-world scenario I would also use unit and integration tests for this particular requirement, which would facilitate error tracing.

R2

- **Approach: Penetration testing**
- **Appropriateness**:
    - This type of testing simulates an attack on the API by a malicious actor, and it can help identify vulnerabilities and weaknesses in the API's security.
- **Limitations:**
    - Limited scope: it is difficult to estimate how aggressive and in what scope a DoS attack might occur. More importantly, the requirement in itself might be flawed: what about other attacks?

        - Justification: This type of testing can still deliver a measure of how robust the API is against DoS attacks (e.g. by iteratively increasing the aggressiveness of the penetration tests.) DoS attacks should be the most likely attack to occur on the API in given application domain.

R3

Testing approach 1:

- **Manual accessibility testing**
- **Appropriateness:**
    - Accessibility testing is a suitable approach for checking whether a UI component can be used with a keyboard for navigation. This type of testing focuses on ensuring that the component can be used by individuals with disabilities, including those who use assistive technologies such as keyboard navigation.
    - During accessibility testing, the tester will use a keyboard to navigate the UI component, ensuring that all functionality is available and that the component can be used without a mouse. The tester will also verify that the component provides appropriate feedback for keyboard actions, such as highlighting the focused element and providing visual indication of the current state. Additionally, the tester will check that the component is compliant with the relevant accessibility standards such as WCAG (Web Content Accessibility Guidelines) and that it follows the best practices on accessibility
- **Limitations:**
    - Limited test coverage: Manual testing is limited to the tester's knowledge, skills and the time allocated for the testing, meaning that it may not cover all the areas that need to be tested.
        - Justification: this is a very specific test scenario (keyboard navigation) which means that the area needing to be tested is limited.
    - Human error: Manual accessibility testing is prone to human error, as it is done by testers who may not have the same level of expertise or knowledge of accessibility guidelines as a specialised accessibility testing tool.

            - Justification: to my knowledge, there is no specialised accessibility testing tool which can simulate keyboard interactions of my system's UI that is not immensely complex to set up. Potential human error is a justifiable trade-off in this case, especially since manual human testing is important to ensure that the component is usable for all users.

Testing approach 2:

- **Automated code scans**
- **Appropriateness:**
    - Cheap but effective way to detect accessibility issues at an early stage.
    - Easy to automate to detect accessibility issues in the component.
- **Limitations:**
    - The effectiveness of this testing approach depends largely on the effectiveness of the tool used to analyse the source code. Depending on the used tool, a high number of false negatives and false positives might make this testing approach questionable.
        - Justification: This limitation can be mitigated by choosing an appropriate accessibility testing tool.
    - Limited to known accessibility issues.
        - Justification: in a real-world scenario, this is something I would address with further testing approaches. It is still valuable to test that an application complies with known accessibility issues as this is a well-research area.

- Choosing two testing approaches for testing this requirement at different levels (UI vs source code) is an example of the **divide principle**.

R4

- **Testing approach: Continuous Development Testing Pipeline**
- **Appropriateness:**
    - If we continuously deploy the system, it will allow us to set up automated tests which check code changes for deployability, and therefore for an aspect of availability.

    - Furthermore, this enables an improved developer experience as errors of certain error classes can be detected automatically and at an early stage.
- **Limitations:**
    - Limited scope: CD testing typically focuses on testing specific scenarios and may not cover all possible cases, this means that issues may still be present in the deployed code.
        - Justification: compilation ability is a specific scenario we are testing.
    - False sense of availability: passing the CD tests can cause developers to have a false sense of availability of the system, especially if they are badly designed. In this case, failure of compilation would detect un-availability, but successful compilation is a somewhat weak indicator for availability. A system may pass the tests and yet still not be available to the end-user.
        - Justification: this is true, which is why in a real-world scenario, I would implement other different testing approaches specifically targeting availability. Furthermore, checking for un-availability is an important part of testing the availability property of a system.

R5

- **Testing approach: top-down functional combinatorial testing**
- **Appropriateness:**
    - There are two parameters, `API route` and `user-status` (authenticated or not authenticated). This allows for a constrained number of combinations.
    - In this case, this is preferable to setting up manual test cases as the tester might leave out important edge cases. Here, we can exhaustively test the entire search space.
    - The top-down integration strategy can be applied early during development if the components are developed in the same hierarchy.
- **Limitations:**
    - Scalability: with added API-routes, the number of combinations will increase and make this testing approach computationally more expensive.
        - Justification: the scale at which the cost will increase is linear as the second parameter is restricted to the two values.

    - Requirement needs clarification as project progresses: which exact API routes should be auth protected?

R6

- **Testing approach: Performance testing**
- **Appropriateness:**
    - Performance testing is a type of testing that aims to measure the speed, scalability, and stability of a system, or system component under a particular workload. If the API can handle a particular workload at the desired speed, this is a good indicator that it is well-integrated with the database. There are other components which are responsible for a seamless communication between system and database, but I have restricted this problem to API (**restriction principle**) which is valid since the API is the entry point of the system backend for database communication, and the database the receiving end.
- **Limitations:**
    - Limited data availability: at the time of testing, the system will not have been deployed in the real world, which means that it is difficult to gauge the scale of real-world user data. We can only estimate the workload at which we want to test the API performance.
        - Justification: since it's only the size that is unknown but we know the structure of the data, we can produce synthetic data at arbitrary scales for this testing approach.
        - However, this does not improve the significance of this test case. (Saying 'the API performs at X kb/s when handling Y amount of data is useful only to a limited amount when not knowing to which relation is stands regarding the real-world scale.')
    - Limited to performance characteristics: Performance testing is limited to performance characteristics such as response time, throughput, and scalability. It does not cover other aspects of the system such as functionality and robustness, which would be other important indicators for checking this requirement.
        - Justification: in a real-world scenario, I would set up further tests to check for other important indicators. However, it is often too 'difficult or not cost-effective to anticipate and completely specify all module

interactions" (Y&T p.406), here those which are involved in the API and database communication. This means that often, a **reduction** of the test approaches and test techniques is needed anyway, so reducing them to performance testing should be validated here.

R7

- **Testing approach: Unit testing**
- **Appropriateness:**
    - We need to test specific method logic at the unit level. Unit testing allows us to test the logic in isolation from other units.
    - This is an example of a testing approach with valid **sensitivity** as we choose a technique that reduces the impact of external factors on the result, and choose a method which is reliable with respect to the property of interest if implemented correctly.
- **Limitations:**
    - Limited to specific tests: Unit testing is limited to specific tests, it may not cover all the scenarios that need to be tested.
        - Justification: since we are given an exact definition in the requirement description, we can assume that the input and test case is well-defined and limited. As the developer of this unit, I know that the number of relevant edge cases is limited and justifies this testing approach.
    - Brittle: Unit tests can be brittle, meaning that a small change in the code can cause many tests to fail. This can make it difficult to identify the source of a failure and can make it harder to maintain the tests over time.
        - Justification: once established as having the correct logic, this unit is unlikely to be revised or restructured as it is a core unit and many other units depend on it. It is more likely that it other units around it may be altered.

R8

- **Testing approach: Performance testing**
- **Appropriateness and limitations:** similar to other requirement with performance approach