GCISLFullStackApp

Project Testing and Acceptance Plan

Sponsor: WSU Granger Cobb Institute for Senior Living (GCISL)

Justin Keanini,

Teni Olugboye

Naomi Dion-Gokan

[10.19.2024]

TABLE OF CONTENTS

I. Introduction

I.1. Project Overview

The GCISL Full Stack App is a web-based system designed to streamline volunteer management and reporting for the Granger Cobb Institute for Senior Living. Its main functions include collecting volunteer information, generating participant reports, and providing an intuitive platform for administrators to view and manage data. The system emphasizes usability and security to ensure data privacy and ease of navigation, meeting the needs of administrators who rely on timely access to volunteer data. To ensure functionality and reliability, this document details a comprehensive testing plan, including both automated and manual testing approaches.

I.2. Test Objectives and Schedule

The primary objective of this testing plan is to ensure that the GCISL Full Stack App meets all functional and nonfunctional requirements outlined by GCISL, verifying that each feature operates as expected, performs consistently across environments, and upholds security standards.
This objective will be met by combining automated and manual testing to thoroughly evaluate functionality, performance, and user experience. Automated testing, including both unit and integration tests, will cover core features such as user registration/login, volunteer reporting, and data retrieval, accounting for both typical and edge cases to catch unexpected issues. Manual testing will focus on usability and functional aspects to confirm that the user interface is intuitive and meets stakeholder expectations. Acceptance testing will be conducted with GCISL stakeholders, confirming that the final product fulfills all requirements and meets both user experience and performance standards.

To support this approach, the project will follow a structured CI/CD pipeline using GitHub Actions, integrated with our Vercel hosting to automate testing, merging, and deployment processes, providing continuous validation of each build. If any issues arise with GitHub Actions or our CI/CD setup, the development team will implement documented manual deployment and testing steps to ensure consistent quality until automated processes are restored.

Testing will proceed in three phases aligned with our development milestones. In Phase 1, beginning in October 2024, unit tests will focus on isolated features, verifying basic functionality for individual components like registration and login to ensure each module performs as expected before integration with other parts of the system. In Phase 2, scheduled for November 2024, integration and system testing will confirm that components interact seamlessly between the front-end and back-end, with special attention to data transfer and security to ensure data flows smoothly and is stored accurately in MongoDB. This phase will also address any performance bottlenecks, particularly in data-intensive features such as the Volunteer Reporting tab. Phase 3 will conclude with acceptance testing in late November 2024, where GCISL stakeholders will assess the app's usability, data accuracy, and overall experience, and stakeholder feedback will guide final revisions to ensure alignment with project goals.

This testing process will result in several deliverables, including a comprehensive suite of automated unit and integration tests covering core features and major data flows, a detailed documentation package for manual tests outlining test procedures, expected outcomes, and observed results, and a CI/CD pipeline configured via GitHub Actions and integrated with Vercel to automate testing and deployments. Should any CI/CD limitations emerge, documented manual deployment procedures will ensure continuity in deployment and testing without disrupting the project timeline.

I.3. Scope

This document outlines the approach and detailed procedures for testing the GCISL Full Stack App. Testing will ensure that all features perform as expected, are user-friendly, and meet stakeholder requirements. Unit, integration, and system tests will validate both functional and non-functional aspects.

II. Testing Strategy

Our testing strategy for the GCISL project will include fully automated tests for core functions, making sure they run smoothly through a Continuous Integration (CI) pipeline. Additionally, we'll use Continuous Delivery (CD) for easier and automated deployment processes. The key areas for testing include user authentication, CRUD operations on tasks and user data, and role-based access control. Non-core functions may also be tested, although they might not be prioritized due to time limits. Our testing process is divided into two main parts: the developer testing process and the playtesting process.

Developer Testing Process

1. Developer Writes Code: The process begins with adding the feature or fixing a bug. The implementation is considered complete when the feature achieves the required functionality.
2. Determine Test Cases: For each core function, at least two test cases will be created: one for expected behavior and one for unexpected or invalid behavior. Non-essential functions might include only test cases for expected behavior.
3. Run Tests: The developer runs all tests, including newly added ones, ensuring the new code doesn't break existing functions.
4. Fix Issues: If any tests fail, the developer finds and fixes the issues, rerunning tests until all pass.
5. Developer Pushes Code to Remote: The developer pushes their code to the remote repository. The code will be run through a CI pipeline, showing the results. Only branches that pass all tests in the CI are allowed to merge.
6. Developer Makes a Pull Request Against Main: The developer creates a pull request against the main branch and adds at least one reviewer. The branch cannot be merged until approved by the reviewer(s).
7. Merge Branch: Once approved, the branch is merged into the main branch. If a Continuous Delivery (CD) pipeline is set up, it will deploy a new release if needed.

Playtesting Process

1. Create Playtest Build: A developer creates a playtest build, either fresh or with modified content to focus on specific test areas.
2. Deliver Build & Allow Time for Testing: The build is delivered, allowing several weeks for thorough testing.

3. Deploy Feedback Forms: Testers are asked to fill out feedback forms about their experience, including enjoyment, learning, and retention.
4. Collect Feedback Data: Feedback is gathered and analyzed for insights on improvements.
5. Make Any Necessary Changes: Based on feedback, necessary changes to content, UI, or user experience are made for the next version.

Not using CI/CD would mean relying on manual testing and deployment, which is time-consuming, prone to errors, and less efficient. For a project aiming for continuous improvement and quick updates, CI/CD is essential. By implementing this testing strategy, we can ensure strong, reliable, and efficient testing and delivery of all project components for the GCISL project.

III. Test Plans

III.1. Unit Testing

The team will use unit testing to validate individual components of the gciConnect system, isolating the smallest units of functionality from other parts of the codebase to ensure that each component works as intended. The following approach will guide unit testing:

IV. **Testing Framework**: We will use the Node js unit test framework for backend functions, focusing on volunteer reporting, data retrieval, and administrator operations. For frontend components, we will use Jest and React Testing Library.
V. **Core Functionalities**: Each core functionality will have unit tests, including functions for adding, updating, deleting volunteer reports, and verifying user authentication.
VI. **Test Coverage**: The team will aim for high code coverage for all core functions, especially focusing on database transactions and API endpoints.
VII. **Testing Sequence**:
VII.1. Developers will write and run tests for newly developed functions before merging code.
VII.2. Tests will be automated in CI/CD pipelines to ensure no core functionality breaks with future updates.
VIII. **Expected Outcome**: All core functions should pass unit tests without errors or unexpected behavior. If any test fails, the responsible developer will refactor the code, rerun the tests, and ensure success before merging

VIII.1. Integration Testing

Integration testing will focus on groups of components, ensuring they work together correctly within gciConnect's architecture. The following steps will guide integration testing:

IX. **Testing Strategy**: Integration tests will be written using Node js unit test for backend testing and Jest for the frontend. Testing will begin with pairing backend components (such as API routes with database operations) and frontend components (UI elements interacting with backend APIs).
X. **Key Integration Points**:

X.1. **Data Sync**: Testing the integration between the frontend forms and the backend APIs that handle volunteer reports.
X.2. **User Authentication and Session Management**: Ensuring that the login sessions and user access levels (volunteers vs. administrators) are managed properly across the system.
XI. **Test Coverage**: The integration tests will cover end-to-end scenarios, such as:
XI.1. Adding a new volunteer report from the frontend and checking its presence in the backend.
XI.2. Ensuring user role permissions restrict or allow specific functionalities as per requirements.
XII. **Testing Sequence**:
XII.1. Integration tests will be automated to run after unit tests, and successful integration will be a criterion for code merging.
XII.2. For each feature update, integration testing will validate the combined functionality of associated components.
XIII. **Expected Outcome**: Integration tests should confirm smooth interactions between components, with no issues in data transfer, user sessions, or permissions. Any failures will lead to a review and refactoring of the affected code

XIII.1. System Testing

System testing will focus on validating the complete gciConnect system as a whole, treating it as a black box to confirm all requirements are met. This includes functional, performance, and user acceptance testing

XIII.1.1. Functional testing:

Functional testing will involve validating the core requirements of gciConnect, focusing on user-facing functionalities:

- **Scope**: Functional requirements include volunteer report tracking, user authentication, report generation, and administrative dashboard functionalities.
- **Test Approach**: Each requirement will have one or more corresponding test cases. For instance:
  - Submit a volunteer report and confirm its presence in the administrator dashboard.
  - Generate a summary report of volunteer activities and verify data accuracy.
- **Testing Sequence**:
  - Each functional requirement will be manually tested by a developer to ensure the system behaves as expected.
  - QA testers will also execute these tests independently to confirm results.

Expected Outcome: The system should meet all functional requirements without errors. In case of issues, the developer will make necessary modifications and re-run tests

.

XIII.1.2. Performance testing:

Performance tests check whether the nonfunctional requirements and additional design goals To ensure that the GCISL application meets its nonfunctional requirements and design goals, our team will employ a combination of backend and frontend performance testing tools. We will use **Postman** for load testing the API endpoints to monitor response times and throughput under various loads. Additionally, **Google Chrome DevTools** will assist in assessing frontend performance metrics, such as page load speed and memory usage.

Since a key non-functional requirement is system reliability during high traffic, **stress testing** will be performed. This involves pushing the API beyond typical usage limits to observe its failure points and determine when performance degradation begins. The primary areas of focus will include latency, CPU, and memory usage, all of which will be monitored throughout these tests.

For any bottlenecks discovered during stress tests, the team will refine the code and database queries to enhance system efficiency and stability. These improvements will be documented and assessed against the initial design goals.

XIII.1.3. User Acceptance Testing:

To ensure the GCISL application aligns with the needs of end-users and satisfies the project's requirements, our team will conduct User Acceptance Testing with selected users who represent the target audience.

A. Resources

- **Working Build**: A stable build of the GCISL application will be provided for UAT participants.
- **Feedback Form**: Users will receive a form to submit their feedback regarding usability, functionality, and overall satisfaction.
- **Developer Access**: Developers will be available to assist users and address any usability issues encountered.

B. Instructions

- **User Selection**: A small, representative group of end-users will be selected to participate in UAT.
- **Testing Process**:
    - Each participant will receive access to the GCISL application along with a brief orientation.
    - Users will interact with core features, such as the user registration, login, and dashboard functionalities, under typical usage scenarios.
- **Feedback Collection**: Users will be prompted to complete a feedback form, detailing their experience, issues encountered, and suggestions for improvement.

C. Revision Process

- **Feedback Review**: The development team will review all feedback and identify common themes or critical issues.
- **Modification and Documentation**: Based on feedback, adjustments will be made to enhance usability and functionality. Any bugs identified will be documented and resolved in the next iteration.
- **Iteration**: If necessary, further testing iterations will be conducted to refine the application until it consistently meets user expectations.

The final iteration of User Acceptance Testing will focus on validating that the GCISL application is user-friendly, meets all functional requirements, and is ready for operational deployment.

XIV. Environment Requirements

Specify both the necessary and desired properties of the test environment. The specification should contain the physical characteristics of the facilities, including the hardware, communications and system software, the mode of usage (for example, stand-alone), and any other software or supplies needed to support the test. Identify special test tools needed.

Answer: Our testing environment is predominantly self-contained, utilizing modern web development tools compatible with both backend and frontend testing. Testing will be conducted through local environments as well as through the GitHub CI/CD pipeline for consistency and efficiency.

For testing purposes, unit and integration tests will utilize Jest and React Testing Library for JavaScript modules and React components, respectively. Additionally, Supertest (Node.js library used for testing HTTP endpoints – for API testing) will be used for testing backend API endpoints to ensure expected interactions with MongoDB.

If available, the team will incorporate GitHub Actions for continuous integration (CI) to automate tests on each commit and merge, guaranteeing that tests are consistently run and passed before changes are integrated into the main branch.

There are no specific hardware requirements.

XV. Glossary

UAT (User Acceptance Testing) Participants: They are real end-users or people who represent the target audience of the application. They help confirm that the system meets the intended requirements and is easy to use.

Postman: it is a popular API testing tool that lets developers send HTTP requests to their backend and check responses, making it easier to validate and debug APIs.

CI/CD (Continuous Integration/Continuous Deployment): A set of practices that automate code integration and deployment processes. CI/CD pipelines run tests automatically upon each code push, ensuring continuous validation of code changes and quick deployment of updates.

Jest: A JavaScript testing framework primarily used for testing React applications. It enables developers to run unit and integration tests on components and functions to verify correctness.

React Testing Library: A tool that enables testing of React components by interacting with the application the way a user would, focusing on component behavior rather than implementation details.

Supertest: A Node.js library for testing HTTP endpoints. It allows for testing API interactions, verifying that backend services function correctly and respond with the expected results.

Unit Testing: A software testing method where individual components or functions of an application are tested in isolation to ensure they work as expected.

Integration Testing: Testing that verifies multiple components of the system work together. In this context, it ensures that front-end elements interact correctly with backend APIs and databases.

System Testing: Testing the entire application as a whole to validate that it meets functional and non-functional requirements, ensuring overall system integrity.

Functional Testing: A type of testing that validates the features and operations of an application, ensuring that they align with specified requirements and that the system behaves as expected in various scenarios.

Performance Testing: Tests that evaluate the application's speed, scalability, and stability under load, focusing on metrics such as response time, memory usage, and throughput.

Stress Testing: A type of performance test where the system is subjected to extreme workloads to identify its breaking point and measure how it handles high traffic or data volumes.

MongoDB: A NoSQL database used for storing and retrieving data in the GCISL application. Known for its scalability and flexibility, MongoDB stores data in JSON-like documents.

XVI. References

Gillis, Alexander S. "What Is User Acceptance Testing (UAT)?: Definition from TechTarget." *Software Quality*, TechTarget, 14 Mar. 2022. [Online]. Available: www.techtarget.com/searchsoftwarequality/definition/user-acceptance-testing-UAT. [Accessed: 02-Nov-2024]

Postman. [Online]. Available: https://www.postman.com/postman/test-examples-in-postman/overview. [Accessed: 02-Nov-2024]