DRAFT FINAL REPORT

SP-27 Full Stack Soulify

Dylan Alexander, Jason Sherres, Jeffrey Cruz

Professor Sharon Perry December 1, 2023

Website

GitHub

Introduction

Soulify is a 3rd Party Spotify Web Application that curates personalized playlists based on “preferred” user activity (i.e., Liked Songs, Listening History, Hidden Songs) & gives users the access to a unique playlist creation tool that mixes music based on tunable attributes.

Test Plan and Test Report

Testing Strategy

For the Soulify SP-27 project, our testing strategy was centered around ensuring seamless Spotify integration, excellent user experience, and stringent data privacy and security measures.

Test Levels

1. **Unit Testing**: Focused on testing individual components for specific functionalities like consent management, playlist creation, user authorization, and database operations.
2. **Integration Testing**: Assessed the interaction between different components, ensuring accurate data retrieval from Spotify, effective communication between the Web UI and backend, and proper user management.
3. **System Testing**: Involved testing the complete software system as a whole to ensure all components worked harmoniously.
4. **Acceptance Testing**: Aimed at verifying the software met stakeholder expectations and provided a user-friendly experience.

Test Types

- **Functional Testing**: Validated system functions according to requirements, covering playlist management, track retrieval, search functionality, and user access.
- **Usability Testing**: Involved real users of varying technology literacy levels to assess the site's navigability and interface interaction.
- **Security Testing**: Ensured robust defense against threats and secure handling of sensitive data.
- **Compatibility Testing**: Tested the application's performance across different browsers and devices to ensure consistent user experience.

Test Cases

Key test cases included:

- **User Authentication**: Testing valid and invalid login scenarios, authorization permissions, and user reauthorization.

- **Playlist Management**: Creating and modifying playlists, ensuring they reflect user preferences.
- **Track Analysis and Search**: Analyzing user top tracks and testing search accuracy for artists.
- **Playlist Creation/Update**: Focusing on user interactions for creating and updating playlists from selected songs.

Non-Functional Testing

- **Performance Testing**: Evaluated the application's response time under various user loads.
- **Stress Testing**: Determined the system's resilience under extreme load conditions.
- **Security Testing**: Assessed the application's capability to withstand security threats and protect user data.

Defect Management

A systematic approach was used to record and manage defects, detailing each defect's ID, title, severity, priority, status, reporter, assignee, report date, expected and actual behavior, along with reproduction steps.

This comprehensive testing approach ensured that the Soulify SP-27 application not only met functional and performance standards but also provided a secure, user-friendly, and reliable experience for its users.

Narrative Discussion

Windows Setup Steps:

Open PowerShell as an administrator to adjust the script execution policy.

Install PyEnv with the provided Invoke-WebRequest command.

Verify PyEnv installation using pyenv help.

Install Python 3.10.0 via PyEnv and set it globally.

Test the Python installation with python -m test.

Clone the repository, create, and activate a new virtual environment.

Install project dependencies using pip install -r requirements.txt.

Mac/Linux Terminal:

Ensure the virtual environment is active.

Set the FLASK_APP and FLASK_ENV environment variables.

Run the Flask application with flask run.

Access the application via http://127.0.0.1:5000/ in a web browser.

PowerShell Session:

Set environment variables for FLASK_APP and FLASK_ENV.

Start the Flask server with the flask run command.

Running Soulify Locally:

For local development, the REDIRECT_URI within config.py must be set to the local server address. This is critical for integrating with Spotify's OAuth 2.0 authorization flow. Care must be taken to switch the REDIRECT_URI to the production URL before committing to the main branch.

Processes and Methods Used

The development of Soulify SP-27 was undertaken using a structured and iterative approach, leveraging modern development practices and tools. Our primary IDE was Visual Studio Code, augmented by several extensions for Python and JavaScript development to improve productivity and code quality.

Version control was a cornerstone of our development process, facilitated by Git with GitKraken as the GUI, offering us a visual representation of our branching and merging strategies. This setup was pivotal in managing our codebase, allowing for a clear overview of our project's progress and facilitating team collaboration.

The Flask web framework was chosen for its simplicity and flexibility, with a focus on a modular design facilitated by Flask Blueprints, allowing us to maintain separation of concerns and improve the maintainability of our code. We adopted venv for virtual environment management to encapsulate our dependencies and ensure consistency across development and production environments.

The configuration of our environment was managed through command-line operations, as depicted in the provided terminal screenshot. These operations included setting environment variables and launching the Flask development server, which we found essential for local testing and debugging.

Challenges Faced and Solutions

Integration with Spotify API: Authenticating and managing sessions with the Spotify API required rigorous testing and refinement to ensure a seamless user experience.

Performance Optimization: The initial version of our application faced performance bottlenecks, particularly in data processing and visualization. Profiling tools helped us identify inefficient code paths, leading to targeted optimizations.

Assumptions Made During the Project

We assumed that our user base would have a basic understanding of music streaming services and familiarity with Spotify. This assumption influenced our design decisions, allowing us to streamline the authentication flow and focus on the core features of playlist creation and track analysis.

Risk Assessments

Risk assessment was an ongoing process. We identified potential security risks associated with handling Spotify tokens and implemented OAuth 2.0 for secure authentication. We also anticipated risks related to scalability and incorporated performance tests to ensure our application could handle increased loads.

Version control practices were a key mitigating factor for operational risks. Our branching strategy, as demonstrated by the GitKraken image, allowed for feature isolation and easier rollback in case of issues. Regular commits and descriptive messages were enforced to maintain a clear development history and facilitate issue tracking.

The software design checklist from the design document guided us in maintaining a high standard of code hygiene and architectural integrity. Tasks such as refactoring the codebase, documenting our work, and setting up deployment automation were all tracked and executed diligently, ensuring a robust and reliable software delivery.

Version Control and Collaboration

Our commitment to robust version control practices was underpinned by the use of GitKraken. This graphical interface allowed our team to visualize and manage the project's branching strategy effectively. The branching model adopted ensured that new features, bug fixes, and enhancements were developed in isolation before being merged into the main branch, minimizing disruptions to the ongoing work and facilitating a continuous integration process.

Regular pull requests and code reviews became part of our routine, ensuring code quality and team knowledge sharing. This collaborative environment fostered by GitKraken and GitHub's remote features allowed for asynchronous teamwork, which was particularly beneficial given the remote nature of our project due to current work-from-home trends.

Continuous Integration and Deployment

Our project embraced the principles of Continuous Integration (CI) and Continuous Deployment (CD) using GitHub Actions. Automating our build and deployment process, we ensured that every commit and pull request triggered a series of automated tests, verifying that new changes did not introduce regressions or break existing functionality.

The CD pipeline was configured to automatically deploy our main branch to the production server after passing all tests, ensuring that our application's latest version was always available to users. This automation extended to our development environment, where local changes could be tested in a production-like setting, enabling us to catch potential deployment issues early in the development cycle.

Design and Development Synergy

The synergy between our design and development teams was vital. Using the images from Soulify's user interface as a reference, our designers and developers worked in tandem to bring to life a user experience that was not only visually appealing but also functionally rich and intuitive. Regular design reviews and usability testing sessions provided critical feedback that informed iterative improvements, ensuring that Soulify remained aligned with user needs and expectations.

Incorporating Feedback and Iterating on the Product

User feedback was instrumental in shaping the evolution of Soulify. We adopted a user-centered design approach, releasing early prototypes to a small user base and gathering insights. This feedback loop allowed us to iterate rapidly on the product, refining features, and adjusting the user interface to better meet user requirements.

Agile Methodology and Workflow

Our development workflow was heavily influenced by Agile methodologies, which emphasize adaptability and fast responses to change. We incorporated several key Agile practices to manage our workflow:

Daily Standup Meetings: These were essential in keeping the team aligned. Each member shared updates on their progress, outlined their goals for the day, and identified any obstacles they faced, fostering transparency and collective problem-solving.

Sprint Planning: We utilized sprints, typically lasting two weeks, to plan and commit to a set of features and tasks that were achievable within the timeframe. This helped us maintain a steady pace and focus on delivering value incrementally.

Sprint Review and Retrospective: Following each sprint, we conducted review meetings to demonstrate the completed work. This was an opportunity for stakeholders to provide immediate feedback. Retrospectives allowed us to reflect on our processes and identify areas for improvement, ensuring continuous refinement of our practices.

Collaborative Version Control with GitKraken

GitKraken played a pivotal role in our version control strategy. Its intuitive interface allowed each team member to work on different branches simultaneously, committing changes without affecting the stability of the main application. This branching strategy empowered us to work on multiple features or fixes concurrently, thereby optimizing our workflow and productivity.

Before merging into the main branch, which would update the live application, we rigorously reviewed changes to ensure quality and coherence with the existing codebase. GitKraken's merge conflict resolution tools were particularly useful in this regard, allowing us to handle overlaps in code changes visually and efficiently.

Integrating Sprint Reviews into Version Control

The sprint review sessions were integrated into our version control practices. At the end of each sprint, we would merge the feature branches into a pre-production branch for staging and testing. This allowed us to review the increments in a controlled environment that closely mirrored the production setup, minimizing the risk of deployment issues.

These sprint reviews were more than just a checkbox in the Agile process; they were a platform for cross-disciplinary collaboration. Team members from design, development, and quality assurance came together to evaluate the application's progress, ensuring that Soulify's growth was balanced and in line with our strategic objectives.

Communication and Collaboration via Discord

To facilitate communication within our distributed team, we utilized Discord, a platform that allowed us to maintain a continuous and open channel of dialogue. We had a dedicated "status" channel where each team member would post their daily objectives, current tasks, and progress updates. This practice is similar to the traditional standup meeting but in a asynchronous format that could be referenced throughout the day.

Discord's voice channels also played a crucial role in our collaborative efforts. They provided a space for real-time voice calls, enabling us to conduct in-depth discussions and screen-sharing sessions. This was particularly useful for complex problem-solving and when visual context was needed. Team members could easily share their screens to showcase what they were working on or to seek assistance on issues they were encountering.

Training with GitKraken

In the early stages of our project, onboarding team members to GitKraken was a priority. Dylan took the lead in training the team on how to use this tool effectively. He conducted walkthrough sessions on creating and switching branches, committing changes, staging updates, and writing comprehensive commit descriptions. These training sessions ensured that even those new to GitKraken could quickly become adept at using its features, which was essential for
Dylan's approach in teaching GitKraken's functionalities was not just about technical know-how; it also focused on instilling good version control practices among the team. Clear and descriptive commit messages, for instance, became a standard, improving the readability of our project's history and making it easier for any team member to understand the evolution of the codebase.

Daily Workflow and Support

Our daily workflow benefitted greatly from the collaborative features of Discord and GitKraken. The "status" channel on Discord became a live log of our activities, offering transparency and accountability. It allowed team members to actively support each other's work, stepping in with help or advice when someone faced challenges. These real-time updates complemented our sprint reviews by providing continuous insights into each team member's contributions.

In addition to the "status" channel, Discord's voice channels were indispensable for our daily operations. They facilitated immediate communication, which was crucial for resolving urgent issues or discussing complex topics that required more than text-based interactions.

Iterative Development and Continuous Improvement

Our development cycle was characterized by a constant push for improvement. Each sprint brought new learnings and opportunities for refinement. Whether it was enhancing the code, streamlining our processes, or improving collaboration, we were committed to evolving continuously.

This iterative approach extended beyond our development practices. We regularly revisited and updated our onboarding processes, ensuring that they remained effective and efficient. As the project progressed, these updates became invaluable, especially when integrating new team members into our workflow.