CS 3300-004 **Software Test Report**

This report reviews the testing scope, test schedule, product features and quality, a list of all flags remaining open with a RISK assessment, and lessons learned.

Test Report Outline

Project Title: Kigo

This report covers testing performed for the following components:

- Front-End Features
- Back-End Functionalities
- SyllableCounter Algorithm and Haiku Generator
- TestingSyllableCounter and compare files
- ProcessLyricLines and RemoveDuplicateLines
- System Integration Testing

The report addresses testing conducted during Increments 1, 2, 3, and 4, including Release 1 and Release 2.

Test team:

Quality Assurance/Testing Lead: Kelly Lowrance, Noah Kabel
Syllable Counter and Haiku Generator Testing: Kelly Lowrance
Front-End Features Testing: Sam Allen

Test Schedule:

a. Unit/Increment testing began 25 October, with revision testing on 01 November
b. Component/Release testing occurred 21 November, with revision testing on 23 November
c. System testing began 21 November

Test Description: Describe each test, what was tested and the test outcome.

---

1. Syllable Counter, Haiku Generator.. (Kelly Lowrance)

Test Descriptions - Kelly
Components Tested:

- SyllableCounter.java
- HaikuFinder.java
- ProcessLyricLines.java
- RemoveDuplicateLines.java

Testing Summary Details:

- **Syllable Counter**: Tested against 1,000+ words, achieving 95% accuracy. Edge cases included irregular words and phrases with triphthongs.
- **Syllable Counter**: Tested against a dictionary list of 69,700+ words, achieving 81.69% accuracy. Edge cases included irregular words and phrases with both diphthongs and triphthongs.
- **Syllable Counter**: Tested against songs which are known for their complex lyrics. Ex: “Rap God.”
- **Haiku Finder**: Successfully generated haikus for the majority of input lyrics, with randomized line selection ensuring variability.
- **Haiku Finder:** Tested against songs with minimal unique lines, numerous back-up vocal lines, and vocable lines. Most difficult aspect of these tests were songs which contained very few unique lyrics.
- **ProcessLyricLines and RemoveDuplicateLines**: Processes raw lyrics to remove duplicates and clean data effectively, maintaining logical integrity across tests. ProcessLyricLines removes back up vocals in parenthesis and non-lexical vocables. Non-lexical vocables are lines consisting of meaningless words repeated, like, “oh, oh, oh, oh.”

SyllableCounter: Testing was incremental at first, beginning with a basic algorithm that counts syllables via counting the vowels in a word. Then the need to account for diphthongs (double vowels that make one syllable) became clear. Incrementally added code to handle special/edge cases like silent “e” at the end of a word. I soon decided that testing would be greatly improved if I wrote a helper class called, “testingSyllableCounter” to test thousands of words at a time. Then evolved into producing a compare file and running the output.txt from syllable count against the compare file which contains the correct syllable count followed by the word (one per line). This also counted the total number of syllables in a file and produced some statistics to show overall accuracy. I then located a dictionary list of over 69,000 english words to test my syllable counter against. I kept tweaking the algorithm until reaching ~82% accuracy before deciding to stop. While this dictionary file is perfect for testing the accuracy of any syllable counter, I decided to look for more lyric-type word files to test against as this was more realistic to our specific application of the algorithm. I ran tests via testingSyllableCounter against a compare file containing the top 1000 most common english words and it scored highly at ~94%.

```
Comparison Summary-
Total words compared: 1000
Matching words: 936
Accuracy: 93.60%
```

Outcomes:

- **Pass Rate**: 81.69%–95% across modules.
- **Issues**: Minor inaccuracies in syllable counting for complex cases.

Lessons Learned:

- Compare files combined with specific testing classes proved quite successful in streamlining the testing process.

---

2. [Front-End Features (Sam Allen)]

Test Descriptions - Sam
Components Tested:

- App.js
- Index.js
- Account.js
- App.css

Testing Summary Details:
Testing for the front end was conducted primarily through live iteration by running local servers for both the React app and the Node.js backend simultaneously. Real-time feedback allowed for rapid identification and resolution of issues. Testing ensured that each feature rendered correctly, functioned as intended, and interacted properly with the backend.

Key Functionalities Tested:
1. Navigation and Button Interactions:
    - Verified that all essential buttons performed their actions as expected (such as the “Connect Spotify” and “Generate Haiku” buttons).
    - Ensured smooth transitions between pages such as the “Home”, “About”, and “Preferences” pages.
2. Spotify Authentication Flow:
    - Tested the redirection to Spotify's login page and the flow back to the app after authentication.
    - Verified that only authenticated users are granted access to protected features like haiku generation.
3. Backend Route Integration:
    - Verified that all backend routes (such as /login, /callback, and /generate-haiku) functioned correctly when accessed from the front end.
    - Tested each route independently to ensure proper handling of requests, responses, and error cases.
    - Ensured compatibility between front-end actions and backend processing.
4. Dynamic Content Rendering:
    - Ensured accurate rendering of user-specific data, such as Spotify usernames, profile images, and personalized haikus.
    - Tested the real-time display of generated haikus, with options to randomly display another haiku.
5. Edge Case Handling:
    - Tested UI response when APIs failed to return data (ex: no lyrics available).
    - Confirmed the display of appropriate error messages for missing or invalid data.
6. Limitations in Error Handling:
    - Due to time constraints, error messages for scenarios where APIs failed to return data were not implemented. While this limitation does not impact the core functionality of the app, providing clear feedback to users during API failures would have improved the user experience significantly.

Outcomes:
- Pass Rate: 95%
- Issues Identified: Occasional misalignment in the UI layout during resizing. These were resolved by adjusting CSS properties and testing responsiveness.

- Overall Status: The front-end features were successfully integrated with the backend and performed as expected.

Lessons Learned:
- Testing with live servers provided an effective way to iterate quickly and address issues in real time.
- Implementing comprehensive error handling, especially for API failures, would significantly enhance the application's robustness and usability.

---

3. [Back-End Functionalities (Noah Kabel)]

Test Descriptions - Noah
Components Tested:
- AuthorizationCodeUri.java
- AuthorizationCode.java
- UserTopTracks.java
- LyricsOvhFetcher.java
- CompileLyrics.java

Testing Summary Details:
Testing validated critical functionalities, including user authentication, Spotify API data requests, and top tracks retrieval, and lyric retrieval. Lyrics fetching achieved 48/50 accuracy for Spotify's Top 50 songs and highlighted limitations for niche tracks due to API constraints.

Key Functionalities Tested:
- AuthorizationCodeUri.java
    - Tested that code provides a valid redirect link to where the user can log into their Spotify account and agree to share any necessary data for the program. This went smoothly without any major issues.
- AuthorizationCode.java
    - Tested that the user authenticated successfully and was able to make requests through the Spotify API relating to user data.
- UserTopTracks.java
    - Tested that code would return a given user’s top songs from spotify. There was one error which would cause the code to return an error then produce the songs immediately after. Error was detected and fixed during the testing stage.
- LyricsOvhFetcher.java
    - Tested that I could retrieve song lyrics using API. Tested against spotify top 50 songs which yielded 48/50. Also tested against my top songs which yielded 9/20. I verified that this was not an error by the code by checking manually if the songs that did not return lyrics had available lyrics on lyrics.ovh. Which they did not. This API is less successful at retrieving more niche, less popular songs. To achieve higher results we could have gone with another API, but this one was free.

Outcomes:

- **Successes**
  - User authentication and pulling data from the user's spotify account was a complete success.
- **Issues**
  - Lyric retrieval was less successful than I had wished for being inconsistent with songs that were not widely popular.
- **Final Status**
  - Functional backend that can successfully retrieve user's top songs and return lyrics for said songs depending on API constraints.

Lessons Learned:

- API's provide an efficient way to collect data and retrieve information without laborious web scraping.
- API's are not perfect and do not always have what you need. If you need better results you will likely have to pay a fee to use a more comprehensive API.

Final product features and quality:

The Kigo app fulfills the core objectives of generating personalized haikus using the users' Spotify listening data. Key features include a user-friendly interface, Spotify authentication, pulling top tracks, lyric retrieval, accurate syllable counting, and haiku generation. The system successfully integrates frontend and backend components with Java libraries for processing logic.

While the project meets all essential functional requirements, there are still some limitations:
Dependency on External APIs

- The Kigo app relies heavily on the Spotify API for user data and the lyrics.ovh API for lyric retrieval. If these APIs experience downtime, rate limiting, or unexpected changes, core functionalities like haiku generation may fail. Additionally, Lyrics.ovh contains a limited database of lyrics and often does not provide lyrics for lesser-known tracks.

Limited Compatibility with Streaming Platforms

- This software is only compatible with the music streaming service Spotify. Users who use other streaming services like Pandora, Apple Music, Soundcloud, or YouTube Music will not be able to use the software.

Error Handling

- Basic error handling and debugging logs are implemented, which facilitate development. However, user-facing error messages are limited and could be improved to better communicate issues, such as missing data, API failures, or unexpected inputs.

Missing feature implementation

- Some planned features (though non-essential), such as the haiku gallery and additional user preferences are not yet implemented.

Overall, the software operates adequately and meets most user expectations by achieving essential functionality.

Test Flags and Risk Management: (List all open flaws/flags remaining and provide a RISK assessment for each)**:**

1. Dependency on External APIs:

Risk: Core functionalities such as haiku generation depend on the Spotify and lyrics.ovh APIs, meaning that downtime, rate limiting, or unexpected changes could disrupt the app's performance. Additionally, lyric.ovh's limited database often fails to provide lyrics for lesser-known tracks, which may harm user satisfaction and reduce haiku variety.
Likelihood: Medium.
Impact: High impact- without these APIs, the app cannot function as intended.
Mitigation:

- Regularly monitor API updates to ensure compatibility with the app.
- Consider other APIs that offer larger lyric databases to improve coverage.

2. Limited Compatibility with Streaming Platforms:

Risk: The Kigo app is exclusively compatible with Spotify, which excludes users that use other music streaming services such as Apple Music, SoundCloud, Pandora, or YouTube Music.
Likelihood: High
Impact: Medium- although Spotify is the most widely used music streaming service in the world, this limitation still restricts the potential user base.
Mitigation:

- Expand the app's compatibility by integrating additional streaming platforms in future versions.
- Provide a fallback option (for example, allowing the user to manually enter track titles).

3. Error Handling:

Risk: Basic error handling and debugging logs are implemented, which facilitate development. However, user-facing error messages are limited and could be improved to better communicate issues, such as missing data, API failures, or unexpected inputs.
Likelihood: Medium.
Impact: Medium- limited user-facing feedback may confuse users or prevent them from troubleshooting effectively.
Mitigation:

- Enhance user-facing error messages to provide clear explanations for issues such as missing lyrics or failed API calls, and how the user should proceed.
- Implement corresponding UI feedback for these error messages, such as pop-ups and alert sounds to notify the user.

4. Missing feature implementation:

Risk: Planned but unimplemented features such as the haiku gallery and additional user preferences reduce the app's overall functionality and user experience.
Likelihood: High for current version.
Impact: Low- these features are non-essential and do not impact the overall functionality of the app. However, the implementation of these features would enhance the app's usability and appeal significantly.
Mitigation:

- Prioritize implementing planned features in future iterations.

- Clearly document which features are currently available and which are planned for future versions of the app.

The identified flaws do not significantly impact the core functionality of the app but may negatively affect user satisfaction and overall usability. These issues should be prioritized in future updates.

Lessons Learned from testing:

- Compare files combined with specific testing classes proved quite successful in streamlining the testing process.
- Testing with live servers provided an effective way to iterate quickly and address issues in real time (For frontend testing).
- Implementing comprehensive error handling, especially for API failures, would significantly enhance the application's robustness and usability.
- API’s provide an efficient way to collect data and retrieve information without laborious web scraping.
- API’s are not perfect and do not always have what you need. If you need better results you will likely have to pay a fee to use a more comprehensive API.