ENSE Capstone Code Review

Soundbyte

06/04/2021

Brandon Clarke, Jirwoun Kim, Mason Lane

1. Table of Contents

Table of Contents **2**

Code Formatting **3**

Architecture **3**

Separation of Concerns 3

Non Functional Requirements **4**

Maintainability 4

Readable 4

Testable 4

Debuggable 4

Configurable 4

Reusability 4

Reliability 5

Extensibility 5

Security 6

Performance 6

Scalability 6

Usability 7

Object-Oriented Analysis and Design Principles **7**

Single Responsibility 7

Open Closed 7

Liskov Substitutability 7

Interface Segregation 7

Dependency Injection 8

Conclusion **8**

2. Code Formatting

- Some code snippets are not following proper naming conventions - however important objects & foundational code are consistent.
    - Some directory names and variables names are not consistent (camelCase, CameCase, and dash between words)
- Our code fits a 14 inch laptop screen. However, some comments go further beyond the screen.
- Most commented code is removed. However, we have several areas in the songMenu/initialization sections that comment out log outputs for debugging. We also have one block of code that has been commented and not removed at the time of this analysis

3. Architecture

3.1. Separation of Concerns

- Our application attempts to follow an MVC pattern. Originally, all business logic would be done via python. Our inclusion of EssentiaJS has made nodeJS part of that business logic. However, our backend functions begin to seep into the front end in certain areas during the initialization steps. The model and view may sometimes be hard to distinguish in certain areas. However, the majority of our heavy lifting is done in a clearly defined backend. Movement of results and data are not as well handled and some of that logic is done at the front end.
- Code follows some code/design patterns especially in the backend/middleware.
    - We use inheritance where we can for like-objects (different forms of suggestion, objects, etc).
- Initial frontend attempts to avoid duplicate code. Later additions to the code base were not as robust.

4. Non Functional Requirements

4.1. Maintainability

4.1.1. Readable

- Middleware and business logic are readable. Initialization is readable & easily communicates how we extract features. Song menu is a very long document. It may be hard to follow.

4.1.2. Testable

- It was hard to test code on python due to the way python worked with the app. We did not build an adequate test environment for the backend.
- Frontend and middleware use debug logs for testing purposes. There was no elegant implementation of automated testing. This made testing components in the middle difficult.

Middleware & initialization could be easily refactored or extended.

4.1.3. Debuggable

- Once again, the lack of automated testing made debugging difficult. However, our console log made it possible.
- We could not easily debug python, as we could not log to our console & lacked automated tests. However, we did test code on jupyter notebooks.

4.1.4. Configurable

- Middleware may allow for a more configurable experience due to the flexibility of song objects. The rest is not easily configurable.
- We might benefit from more, clearly defined constants
- Some constants are unnecessary.

4.2. Reusability

- We repeat our code in the frontend.
  - Few generic functions - although listupSongs has been modified to accommodate a variety of unique situations
  - CSS is not standardized; we have several areas where CSS is duplicated.
- Middleware & Initialization attempts to be flexible.

    - The suggestion object may be grown to include other forms of input/output in its inheritance
    - So may the various types objects
    - Many generic classes. Few generic functions
- Python has duplicate code for each form of suggestion. However, we could easily make a single python script reusable with some adjustments in nodeJS.

4.3. Reliability

There is some exception handling among the initialization & song menus. It is not fully tested & there are areas that could be of concern. For instance, improperly formatted JSON files. We have exceptions for missing files - but may not have such for poor formatting. Furthermore, users are prevented from inputting certain values in the feature input section.

We also have poor exception handling for poorly formatted WAV files - however we can reset and restart the entire process in some cases.

Python has semi-reliable exception handling. It does not break the application - but resolving an error is not properly handled. Errors are posted on the debug log, and the user is not informed. The user must return to the previous menu themselves. We could easily add exceptions, as our python-shell will either return an Song object or throw an Error

4.4. Extensibility

- Python-shell makes replacing components for our py scripts fairly easy. Simply change the python script being used. We could easily test different algorithms
- The use of TypeScript as the main language of our application affords us a structure of object-oriented types along with the freedom of Javascript. This allows us to quickly add new features as the majority of the functionality is already inherently present within our types. To extend the scope of our app would only require a few slight modifications since the code is designed to be reused.
- Middleware may be grown with different suggestion components due to the simple suggestion interface class.
    - Types may also be grown.
- Frontend is hard to grow - we must refactor the song menu as it has bloated functions. Adding features requires extensive changes to the code within songMenu.js

4.5. Security

- Implementation of content security policies
    - Using unsafe eval for our WASM is not best practice
- User input is limited with the use of checkboxes and field sizes (feature input). However, we have not done testing to see how a user might modify the HTML elements to bypass these forms of input.

4.6. Performance

- JS checks the data type when it executes and can not benefit from the performance improvements afforded by TS. However, our JS does rely on TS files which greatly improve the handling of data within JS.
- Performs well in every category minus the initialization step. Using a JSON allows for quick access/analysis of extracted song data.
- Initialization step is asynchronous and is a lazy loading solution at the moment. Future iterations will leverage our web workers to propage songs synchronously. This is better practice than what we are currently doing.
    - Unhandled WASM memory failure with 200+ songs. We may fix this bug as well as the long load times with synchronous processing of songs.
- We store session data within a JSON to avoid unnecessary re-initialization (avoid long load times).

4.7. Scalability

The amount of users that our application can support is only limited by the amount of users that we can distribute this software to. This is due to the fact that our application runs on the users computer and requires only a library of songs.

This is technically a chromium web-app, and may be used with other frameworks aside from Electron to reach other devices like phones.

The amount of songs allowed in the users library is theoretically infinite though a better queuing system for extracting song features would need to be implemented. Currently the application will run out of memory due to processing ~200 songs at once.

4.8. Usability

Usable in every area except for the initialization. Wait times are too long & are not an enjoyable experience. This is why we are putting so much emphasis on our synchronous web-worker proposal.

5. Object-Oriented Analysis and Design Principles

5.1. Single Responsibility

- Middleware including types & suggestions follow a single responsibility principle and use inheritance to evolve a type into new responsibilities.
- SongMenu handles song lists & result lists. This could instead be split into two different processes.
- Python scripts follow a single responsibility principle but achieve this with duplicate code.

5.2. Open Closed

- Middleware is open to extension and closed to modification. New types or suggestion models may be evolved from existing classes.
- Python scripts may be easily swapped out for experimentation, following the open closed principle.
- Additions to the frontend songMenu usually require other changes to be made with the existing code. It is not following the open closed principle.

5.3. Liskov Substitutability

Middleware does not change the behavior of the parent class. Suggestions always produce results - but suggestion children may accept different forms of input to accept that result. See suggestionWFeature and suggestionWSong.

5.4. Interface Segregation

- Interfaces are not lengthy and achieve specific object-oriented purposes. Some interfaces in the middleware have dependencies.

5.5. Dependency Injection

- Most objects do not have hard coded dependencies. However, libraryData does include one hardcoded dependency.

6. Conclusion

Our application starts with a healthy initial foundation, however it is very clear where we began to add code hastily. Much of our application is still object oriented, open-closed, scalable & follows the single responsibility principle. However, the frontend should be refactored to further take advantage of the established TS objects/architecture. Our frontend should avoid certain backend/middleware responsibilities. Song lists and results lists should be split into their own scripts, and functions should be simplified & extracted from our current code. By making generic functions, we may save a lot of confusion, making the code more readable & extendable.

Initialization frontend and backend should be redesigned to avoid the lazy loading currently implemented by our feature extractor. During this time is the perfect opportunity to refactor & redesign the frontend. With the inclusion of synchronous song propagation & refactored song/result list displays we may also implement more effective exception handling. This next iteration may also allow for the creation of more effective testing - including automated tests.