Spark Collaborative Editor Final Report

Jeremy Dormitzer

Abstract

For my senior project, I decided to build a collaborative text editor in the style of Google Docs. I hoped to make a tool that could support remote pair programming, in the classroom and in the workplace. After drafting an ambitious plan, I pared down the scope of the project to a more realistic level. The final result was a basic web-based text editor with some advanced features and full support for collaborative editing of documents. Although the software I built was not as fully-featured as I had hoped, I considerably advanced my software engineering skills and built a basic product that could be built upon in the future.

Introduction

For my senior project, I built a web-based collaborative text editor called Spark. My decision to build a pair programming tool was motivated by my experience collaborating with other students on school projects. Although version control tools like `git` or `mercurial` enable collaboration on the project level, they do not offer the ability to collaborate in real time on a single file. Existing tools for real-time collaboration do exist (see the Background section for more details); however, these tools have various problems that make them unsuitable or difficult to use. Another major motivation for building Spark was my interest in the open source software movement. I wanted to build a tool that could be developed collectively by an open source community. This influenced my choice of technology - by choosing an all-JavaScript stack, I hoped that more developers would be able to contribute to the project. Unfortunately, I did not end up having time to properly pursue community outreach or repository maintenance (writing a contributing guide, good internal documentation, etc.), so I had to abandon that aspect of the project.

Background

Collaborative real-time editing on the web is not a new idea. In 2005, a company called Upstartle released Writely, an online word processor that supported real-time collaboration. Writely caused a media buzz, and was later acquired by Google and rebranded as Google Docs, a collaborative rich text editor that remains at the top of the web-based word processing market. However, Google Docs does not have any code editing features, such as syntax highlighting, autocompletion, or error checking. More recently, some companies have built

complete web-based IDEs (integrated development environments). These projects include syntax highlighting and other code editing features. Some even give users web access to a VPS (virtual private server) in which they can compile, test, and run their code. These editors include Squad, Firepad, and Cloud9, among others. When I initially researched this topic, I thought that all existing solutions were proprietary, i.e. closed source, but further research revealed that Firepad is in fact open source. As it turns out, Firepad is pretty similar to the project that I wanted to build, although it was meant to be embedded rather than running as a standalone application.

The application that I wanted to build would have stood out from existing projects (perhaps other than Firepad) due to its open-source community and its focus on pair programming. However, because I did not achieve everything I had outlined, Spark is instead in a position where it can draw influence and inspiration from other web-based text editors.

Methods

In my original proposal, I split the list of features for the project into two categories: MVP (minimum viable product) and stretch features. I planned to fully implement all MVP features and implement stretch features if I had time. Here is the breakdown of both lists:

MVP Features

- Real-time collaborative editing
- Basic security
- HTTP Basic Authentication
- Encrypted web socket traffic
- Autosave
- Syntax highlighting (using an existing solution)
- File tree
- Text chat
- Tabs and multiple open files
- Configurable via the UI or a config file

Stretch Features

- Plugin system (for things like third-party integrations or debugging support)
- Moveable, resizable panes
- IDE Features
  - Autocompletion
  - Refactoring
- Video/voice chat
- Multiple key binding schemes
- Color themes

- Git/Github integration
- Support for more languages
- Progressive web app for easy mobile/tablet use
- Desktop client built with Electron

Certain technical decisions I made (detailed below) enabled me to achieve many of the stretch features. Conversely, some of the MVP features ended up being too complicated to implement and were left out. Here is a list of features that got implemented:

Implemented Features

- Real-time collaborative editing
- Authentication
- Syntax highlighting
- File tree
- Autocompletion
- Error checking
- Support for most programming languages

From the list, it is clear that my original plan was too ambitious. Despite this, I believe that I accomplished a significant core portion of the idea.

Spark consists of two parts: the server and the web application. The server is responsible for keeping track of the state of all files being edited, as well as authenticating and keeping track of connected users. The server holds the "master copy" of each file. The web application is the actual editor UI. It connects to the server via WebSockets to enable close-to-real-time networking. The web application allowed users to create a new account, log in, create a new project, open an existing project, edit files in a project and save changes, and create new files and directories in a project. Due to time constraints, I did not build an admin interface to the server - instead, the system administrator maintaining the Spark server instance could perform administrative tasks like adding new users, changing user permissions, etc., through a CLI (command-line interface) that ships with the server. This also meant that there was no way to add a user to an existing project from the web application - that could only be done via the server CLI.

I needed an efficient solution to keep clients synchronized as they edited the same file. The naive approach would be to send the whole file contents to the server every time the user made a change, and to broadcast that data to every other client editing the same file. Of course, sending the whole file contents over the network is monstrously inefficient. Instead, I simply recorded each change that a user made and sent them one at a time over the network. They were replayed on the server, and broadcast out to each other client to be replayed on their copies of the file. This approach is much faster, and handles multiple clients editing a file simultaneously quite well. The problem with this solution was that clients could get out of sync with the server or with each other if they received the changes in the wrong order, which was entirely possible due to network latency.

To solve this problem, I made each client periodically sync up its buffer contents with the master copy on the server. This did involve sending the whole file over the network, but the performance penalty was much less because the sync only happened periodically.

I chose a tech stack that I was familiar with, for the most part. I wanted to focus on the interesting technical and algorithmic challenges of building a collaborative text editor without having to worry about learning any new technologies on top of that. Also, as mentioned previously, I wanted to use technology that many other developers were familiar with. For these reasons, I wrote the server with Node.js. JavaScript is very familiar to me and many other developers. Another reason I used Node was its excellent WebSocket support via the Socket.io library. Although this library is also available for Python and other languages, it was originally written in JavaScript and the Node version is the most stable and easy to use. The server used WebSocket events to communicate everything to the client - file updates, authentication, creating new accounts, and all the other networked behaviors. I used a SQLite database to store user information and project metadata. I did not feel the need for a heavier database solution because most of the important state of the server (i.e., the text in the files it tracks) is stored either in memory while the user edits a file or in the file system when the user saves a file.

On the client side, I chose to use the Vue.js JavaScript framework to build the UI. Vue is a framework which binds DOM elements to data, and automatically and efficiently updates the DOM when the data changes. This makes it trivial to create highly interactive, data-rich web applications such as a code editor. Vue applications are built from components, which are reusable elements that take in some data and render it to HTML. Using components allowed for an extremely modular and DRY ("Don't Repeat Yourself") architecture. To implement the actual text editing, I used the Ace editor library. This turned out to be an excellent decision, as I mentioned above, because Ace is extremely powerful. It gave me syntax highlighting for most languages, autocompletion, and error checking basically out of the box. The flip side, however, is that the documentation for the library is terrible. I had to look up how to do almost everything on Stack Overflow, and even then there was a fair amount of debugging, reading source code, and trial-and-error involved in getting the thing to work. Despite those hiccups, it was an excellent choice because it made the final product much more polished than it otherwise would have.

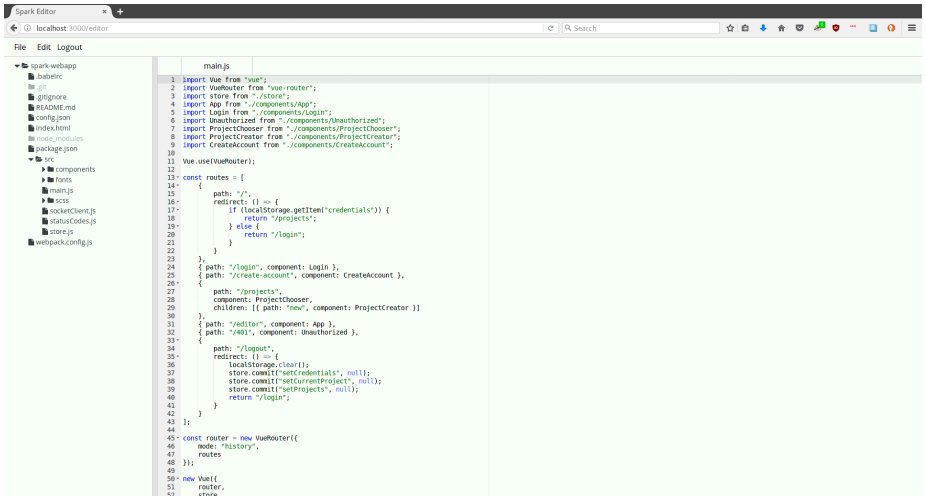
Results

The editor turned out quite nicely, even though it was not as fully featured as I wanted it to be.

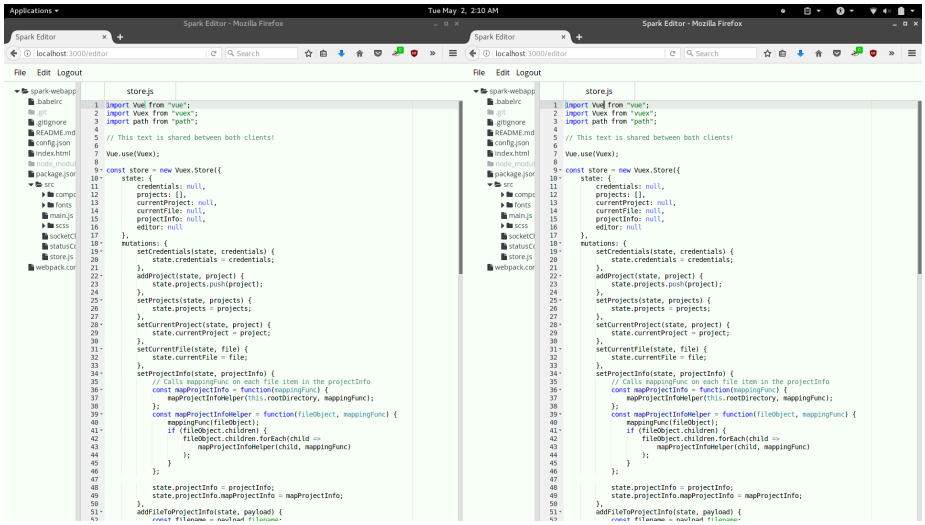
The functionality of the final editor is detailed above.

Screenshots

Main Window



Collaborative Editing



Lessons Learned

In building Spark, I learned quite a bit about software architecture and design, networked applications, and realistic project scoping. Vue helped me see the benefits of modular code, to the point where I could write reusable components that were totally generic (i.e., all data used to render them was passed in from

the outside). I even built some recursive components like the file tree - a file tree is really a recursive tree where the nodes are directories or files. Building an application that is meant to be used over the network means factoring in different concerns than those involved with writing local-only applications. For example, network latency meant that certain changes were non-deterministic - changes to files from other clients could arrive in a different order than they were sent. In coming up with solutions for these types of problems, I learned to build failure-resistant systems and algorithms. Finally, I learned that my idea of a reasonable scope for a project is usually way off. Next time I plan something this big, I will decide what features I want to include, then cut the list in half (or even smaller). Software projects, even smallish ones, inevitably involve complications and unexpected concerns. I learned to plan for that.

Conclusions

Although the final version of Spark was nowhere near my initial plan, I am proud to let it represent the culmination of my CS degree. Through it, I demonstrated good software design principles and showed proficiency across all levels of the stack, from the database to the server to the web application. That being said, there are a few changes I would still like to make, and hope to make them once I have free time again. First, I would like to polish the UI and add missing features. These include the ability to add collaborators to a project, the ability to delete files or directories, being able to see whether a file is saved or not, and being able to customize aspects of the application such as keybindings or color schemes. I would also like to refactor the server to allow projects to be run in their own containers or VMs on the server machine. This would allow projects to be compiles or run in a safe, sandboxed environment. To make this happen I would also need to add some sort of web SSH interface to the editor. Finally, and most importantly, I would like be build an open-source community around the project. This was one of my original goals, and I regret not leaving enough time to make it happen. This would entail doing PR outreach for Spark, writing better documentation, and making myself available via GitHub or other channels to respond to requests or issues. Achieving these goals would let me feel that Spark is complete.

References

Writely word processor: https://en.wikipedia.org/wiki/Google_Docs,_Sheets_and_Slides

Squad editor: https://squadedit.com

Firepad editor: https://firepad.io

Cloud9 IDE: https://c9.io

Vue.js: https://vuejs.org

Spark editor proposal: https://jdormit.github.io/senior-project-site