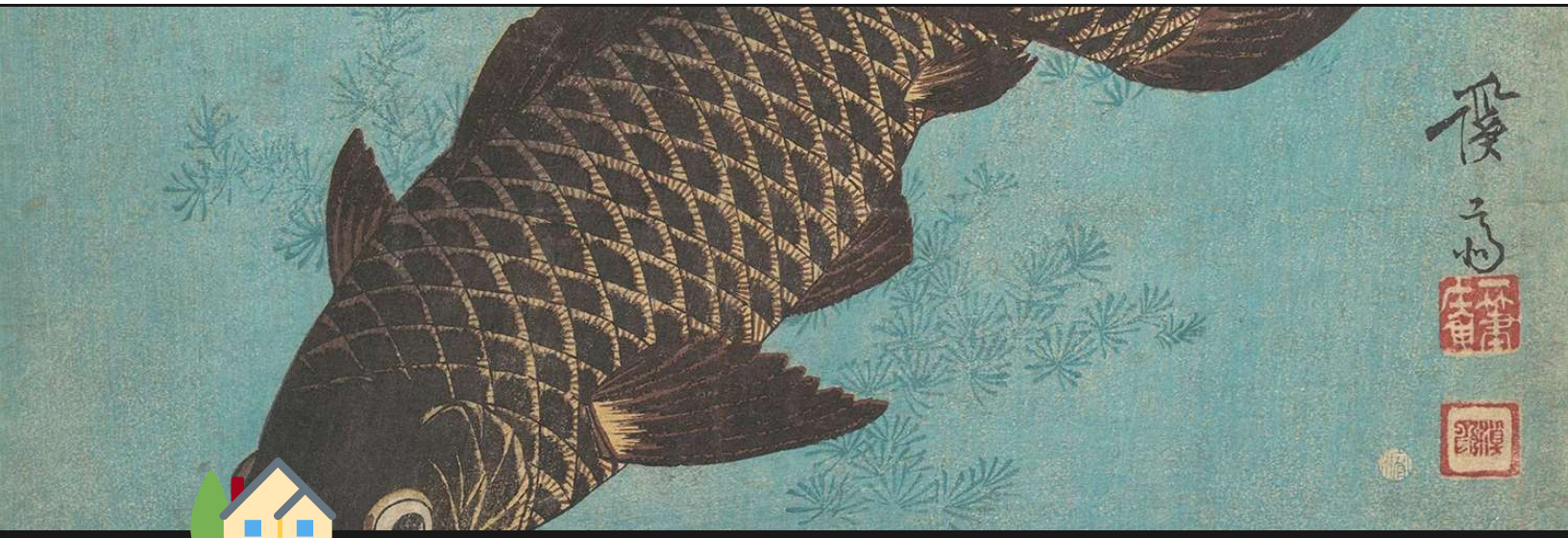

Cadmus Software Engineer Homework Task (Platform)

Owner	Rishi Ranjan
Tags	Empty
Date	Empty

2 more properties

We have created a Homework Coding challenge that requires a solution in React + Simple Backend API (Typescript and NodeJS). It involves building **features in a full-stack web application.**

We expect the entire **Task to c**omplete at most **4 hours** of total work. *You can start whenever, and the timing is just a suggestion.* Let us know if you are pressed for time, and we can always accommodate you.

You will be evaluated on the **Success Criteria** listed in each of the Tasks and your Coding design and architecture. You can use any React and backend ecosystem libraries to supplement your architecture and data flow. Our evaluation rubric includes:

- Attention to Detail - UI + UX, Code style, handling of edge cases.
- Clarity of implementation - Logic flow, Readability, Functional code.

- Browser API knowledge.
- Data Structures.
- React best-practices.

Once you implement the **criteria of the 3 Features**, you should archive your project (**without** `node_modules` **or build dependencies; use** `.gitignore`) and send it back to us.

Please email us early for assistance in starting or progressing through the task. You should email the finished project archive back to us via an email to `rishi@cadmus.io`

Good luck! We are eager to see what you create 😄

> ⚠️ Please note that you cannot upload this task in a publicly visible repository or host. The artefacts and code used to set up this project are private.

Welcome to the Cadmus Homework Task for full-stack developers.

You will create a Typescript + React Single-Page-App (SPA) front-end client and a simple API backend to create a complete application with **3 Features**.

Development Requirements

For this task, you will create a single project with the `client/` and `server/` source code.

CLIENT - Typescript + React

The `client/` codebase is required to be a **Typescript** and **React** codebase producing a **Single Page Application**. You do not need to build this codebase as a static production asset, the development server will suffice.

We recommend using simple front-end project bootstrapping tooling like:

1. Vite OR
2. Create React App (CRA)
3. Or something straightforward

Please keep it simple to run with instructions in the README or package.json scripts.

SERVER - Typescript / Elixir / Go / Rust recommended

You can write the `server/` codebase in **the above-recommended languages.** Please keep it simple to understand and use the minimum subset of web-server libraries required to serve the **Backend API using HTTP or WebSockets.** We will evaluate your fundamental understanding of web servers and creating APIs. The `client/` codebase will interact with this API.

Please provide a README with instructions on how to run the `server/` code.

During development and evaluation, both the client development server and the backend server will be run on the same host on different ports. For example, the React dev server will run on `http://localhost:3000` and invoke the Backend API on `http://localhost:4000`. You may do so if you can serve the React SPA through your web server.

Application Overview

The complete application (created by you) has a web-based editor. This text editor should be built using either:

1. DraftJS React editor framework **OR**
2. TipTap (ProseMirror-based) editor framework.

A User of this application should be able to use this **full-page editor** to write a long-form essay document. They should also see a **live word counter**. Their **document versions should be saved** regularly to the server while ensuring it is not hammered with save requests on every change. The user will also see a **save status indicator.** The saved document should be loaded into the Editor on refresh. This is the basic skeleton of an editing environment like Notion or Google Docs. You can take inspiration from both. **[FEATURE-1]**

Next, you will add a session-locking feature that prevents editing the **document** simultaneously from multiple tabs, users, computers, etc. Only one document exists, but the saved history should always follow a linear timeline. The goal is only to allow edits on the latest version of the **document.** When a session tries to make saves on a stale version, they are asked to re-fetch the latest changes before proceeding. [**FEATURE-2]**

The next feature requires comparing every **document** save version with its previous version and computing the total ***words added***, **the** ***words removed***, **and the** ***words common*** between the current and previous versions. The comparison is on a simple word list level in the backend server. This feature aims to create an API endpoint that returns a chronological sequence of word count metrics for the document's lifetime. There is no client-side change required. [**FEATURE-3]**

Features

FEATURE-1 - Online Editing Environment

In implementing this feature, you will set up the primary user interface for the application. You can implement your view as you see fit and choose your design. Please take inspiration from existing online editing environments like Notion and Google Docs. The 4 main components that need to exist in your application after this feature are:

1. Text editor (Rich Text or Plain Text) to write essay-type documents.
2. Live document word count rendered in the interface.
3. Auto-save for the document with a backend API server.
4. Document save status indicator

Text Editor

The requirement for this editor is to provide the user with enough writable area to write long-form essays. Implementing simple Rich-Text capabilities in your Editor or a ToolBar is not required but would look nice. Use either of the recommended libraries of Draft-JS or TipTap (ProseMirror). You can design your environment to your choice.

Word Counter

The user should be able to see the live document word count in the UI. How a word is defined should make sense for paragraphs in English. It should be consistent and should handle trivial cases.

Saving and Loading

The document should be regularly saved to the backend server. On a refresh, the latest content should be available in the editor. You can save the entire document as a version or only the changes/operations.

The frequency of the saves should be optimised to not fire on every change. Assume the web server has a rate limit; therefore, the client reduces the frequency and size of save requests.

The only storage you need to use on the server is the memory. **You don't have to use persistent storage like a database to store them. The document contents should only be preserved between client refreshes and NOT server restarts. There is only 1 user and 1 document in this task.**

Save status indicator

The user should see a saving status indicator that covers the saving, saved, and error states of the saving network process.

✅ SUCCESS CRITERIA

- ☐ **[Required]** The Editor on the page can write a complete essay.
- ☐ **[Required]** Updating word count is displayed somewhere in the UI
- ☐ **[Required]** Saving editor contents to the backend and loading on a refresh
- ☐ **[Required]** Save background process is optimised to not trigger on every editor change
- ☐ [**Required**] Save process status indicator somewhere in the UI

FEATURE-2 - Linear Timeline

In this next feature, you will ensure a single document follows a linear global order regardless of how many client sessions have that document open. You will add countermeasures against editing stale document versions across multiple tabs or simultaneous sessions. Changes made on a non-latest (not HEAD) document version should be **prevented from saving** or **mitigated completely.**

Only one editing session should be allowed to change the document's latest state (HEAD) at any given point, even if multiple tabs are open. The UX and backend communication implementation is up to you.

Note that you are not implementing collaborative editing here but are implementing Multi-Reader Single-Writer (MRSW). This is a simplified scenario of an entire collaborative editing environment.

💡 **Sample Scenarios:**

1. A document in two tabs or two different browsers. They make edits in tab A and close it. They then switch to tab B and make edits. Without your implementation, the edits on tab B will be on an older version. If the user doesn't realise this, they will lose the edits they made in tab A.
2. A document is being edited in 1 tab. Another tab is opened to edit the document. Your feature should ensure only one of these tabs saves their changes and the other is stopped and notified. The choice of the tab is up to you.

✅ **SUCCESS CRITERIA**

- ☐ ***[Required]*** Full-Stack locking solution to prevent multiple writers but allow multiple readers
- ☐ [Optional] All clients get the latest content reactively

FEATURE-3 - Word Change Analytics API

Now that your application is useable as an online editing environment, it will need an analytics endpoint. You will add an API endpoint to the server which returns a chronological version history of the document with word change metrics (between each subsequent save) as such:

```
GET /wordcounts HTTP/1.1 200 OK Content-Type: application/json [ { "version": 0, "added": 1, "removed": 0, "common": 0, "content": "Hello" }, { "version": 1, "added": 1, "removed": 0, "common": 1, "content": "Hello World"}, { "version": 2, "added": 3, "removed": 0, "common": 2, "content": "Hello World, How are you?"}, { "version": 3, "added": 0, "removed": 2, "common": 3, "content": "How are you?"}, { "version": 4, "added": 1, "removed": 1, "common": 2, "content": "How are we??"}, { "version": 5, "added": 0, "removed": 0, "common": 3, "content": "We are how"}, { "version": 6, "added": 3, "removed": 0, "common": 3, "content": "We are how We are how"}, ]
```

Each JSON metric entry in the response is a Word level comparison (each save is tokenised as simple words) of the incoming document save version and its immediate preceding version. In other words, you are computing *how many words were typed, how many existing words were removed,* and *how many words stayed common* on a save to get to the new save.

The calculation must happen on the server. We are building a replica of a data pipeline which may need to more heavier computation asynchronously in production. You should also consider when the metric is computed and how that decision will affect real-world production load.

SUCCESS CRITERIA

- ☐ **[Required]** API Endpoint to *GET* word added metrics for every to save in sequence
- ☐ [Optional] Precompute the word metrics and store it in an in-memory structure as saves come in
- ☐ [Optional] Display the word difference on the front end on every save beside the word count

---

Submitting the Homework Task

After you have finished making and testing the changes and features to this project for the 3 Tasks, you can archive the entire folder (**without any dependency and build artefacts like node_modules**) and send it back to us via email.

> **REMEMBER**
> Your codebase will be a Typescript React SPA with a *server.ts* file. Please provide instructions in the README on how to run both the server and client development servers.

If you are using git to track your work, you can create an archive of the tracked files only via the `git archive` command (assuming you are on the branch `main`):

```
git archive --format=tar.gz -o cadmus-homework-fullstack.tar.gz main
```