USDS Regulatory Analysis Web Application

Project Documentation

By David Morris

September 22nd, 2025

Introduction

Greetings. My name is David Morris.

I wish to thank the United States Digital Services for this exercise. As a full stack developer, I figured that for the sake of a quick turn around time, we should rely on springboot and hibernate for the backend with angular on the frontend.

Of course, I'm up to speed with agentic AI as a form of assistance and I continue to research deeper on that direction every day (via many resources such as deeplearning.ai, azure ai course, local meetups, etc), but I won't digress too much on that here.

To get warmed up, on the day I received the assignment (Friday) I just allowed Claude Sonnet 4 to do whatever it wanted just to see what kind of crazy hallucinations it'd come up with. That way I'd better understand where to direct my *actual human effort because wow do those hallucinations stack up!*

Given what time I had in my schedule over this weekend I took to better understand the ecrf.gov and it's api instructions.

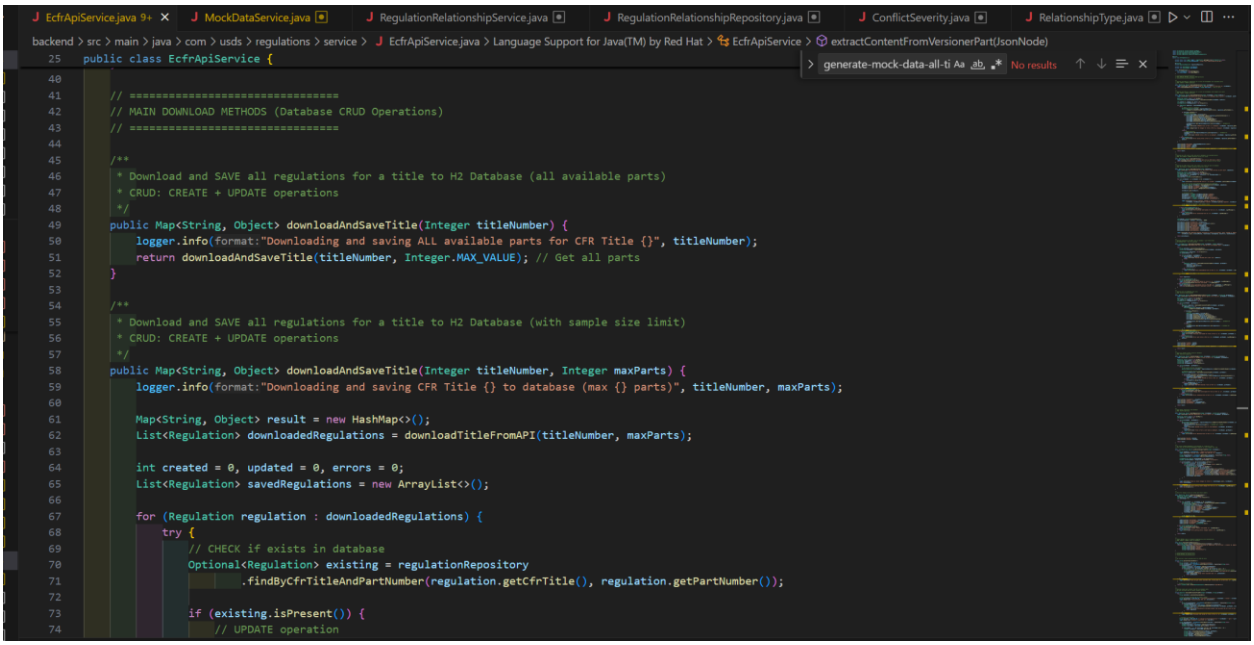
All that considered, let us just count what I did today here on Monday the 22nd since it's less of a prototype and I was able to have a clear schedule to put my full professional attention. If you're curious about the crazy pure hallucination version of my code, here's the link to that repository, but consider what's here to be the real submission.

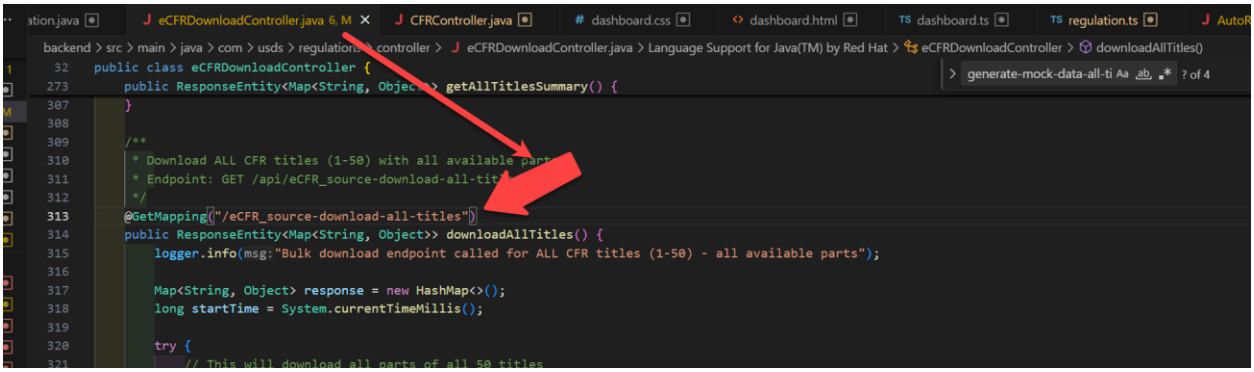
So, having started fresh at roughly 10 am as of the day of this writing, let's see how things went.

Backend Documentation

Started at 10 am.

After initial SpringBoot scaffolding on localhost:8081, the first step begins with our *EcfrApiService.java*, which is responsible for downloading data in accordance with the api endpoints found at https://www.ecfr.gov/api. CRUD controller services such as the *eCFRDownloadController.java* class would then handle translating these pulled requests into my database using *Regulation.java* as our entity class.



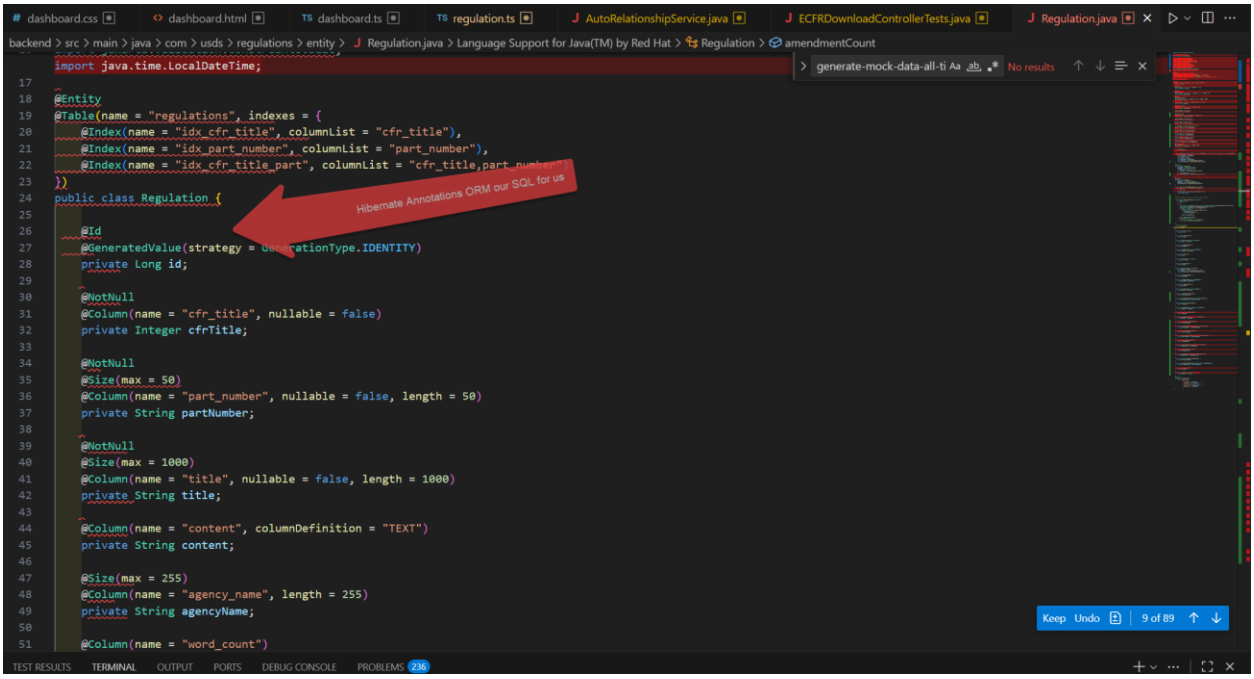


I was very interested in this phase on how I could download and save tiles in any number of ways (hence method overloading), including how to do it in a manner wherein I can define a proper sample size. Afterall, downloading and pushing to a database where we have 50 titles and up to 200,000 documents could take hours of testing time and we certainly aren't interested in that!

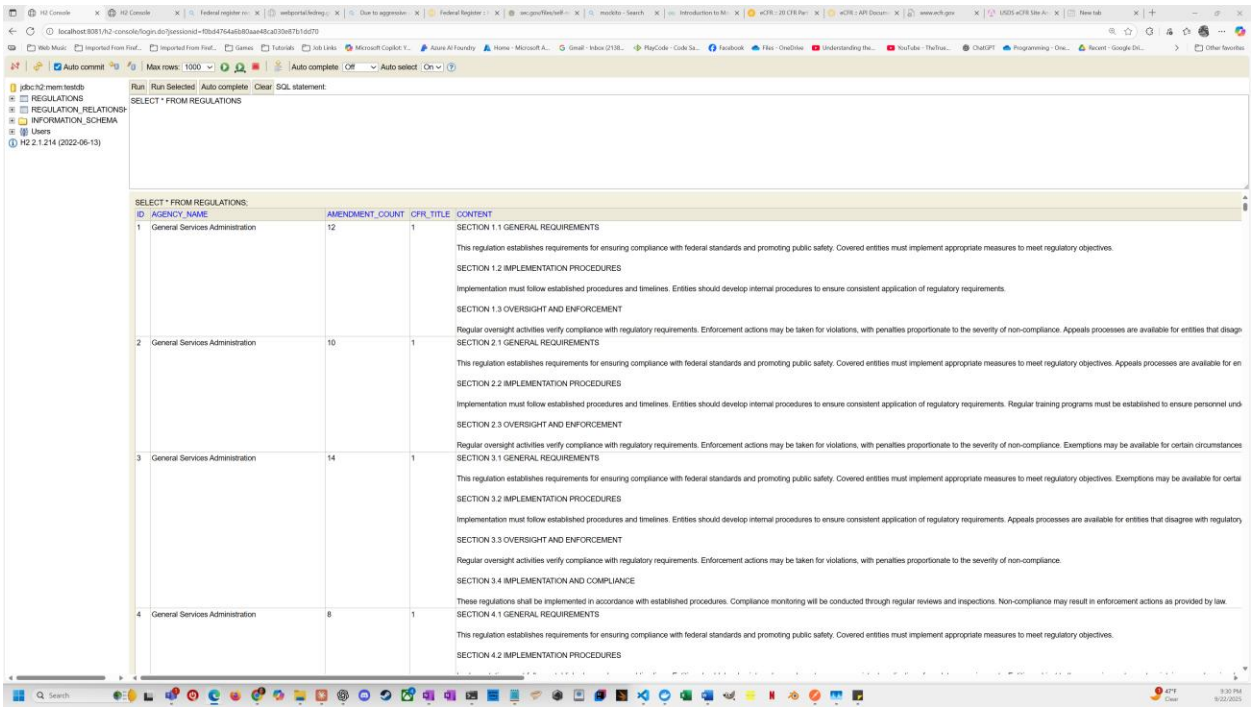
The *ecfrApiService.java* class is called by our *eCFRDownloadController* to take these results and parse them into a database schema that I'm letting Hibernate annotations handle as *Entity* objects.

Schema Definitions - Hibernate

First model object was the *Regulations* object to capture everything we'd need for the core metric objectives.

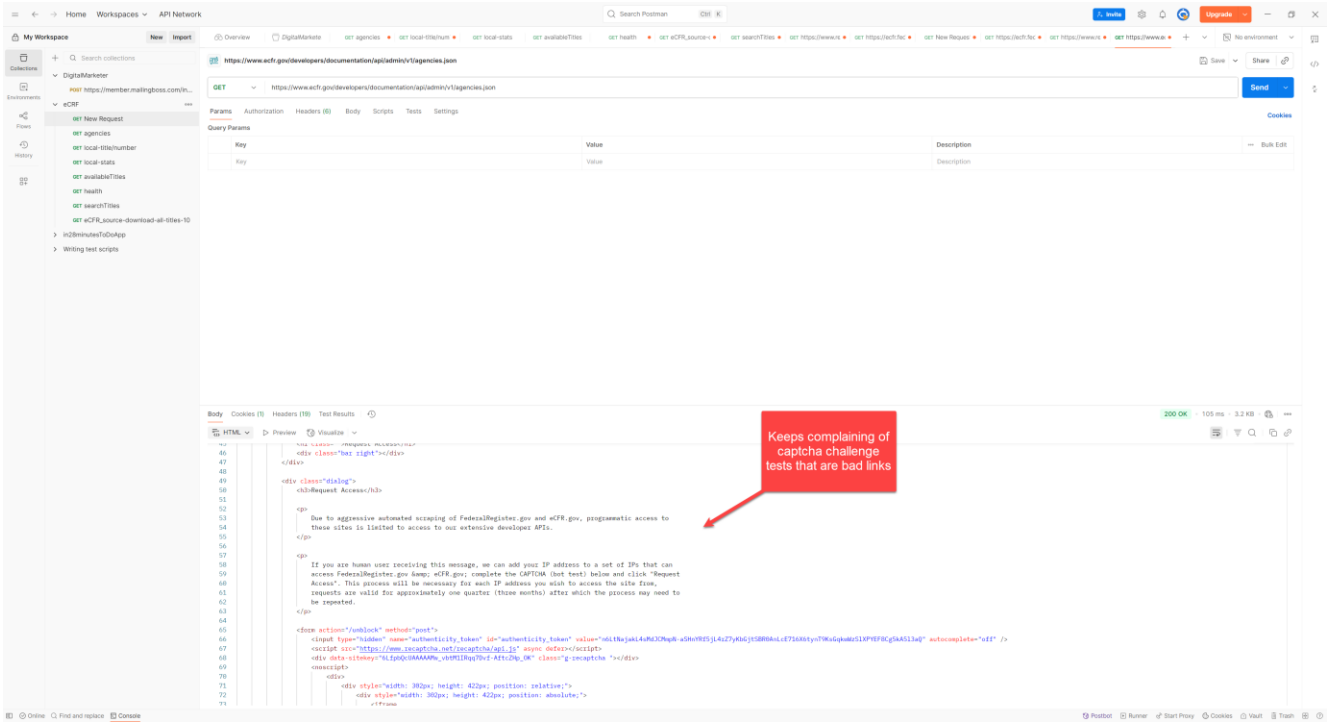


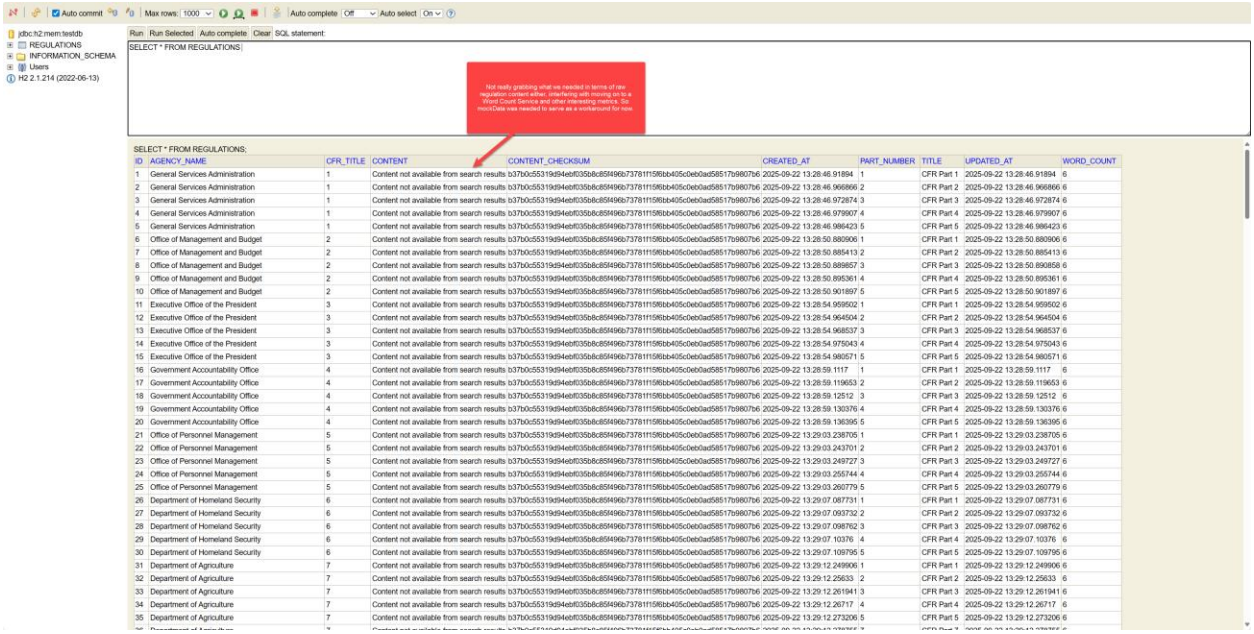

```java
import java.time.LocalDateTime;

@Entity
@Table(name = "regulations", indexes = {
    @Index(name = "idx_cfr_title", columnList = "cfr_title"),
    @Index(name = "idx_part_number", columnList = "part_number"),
    @Index(name = "idx_cfr_title_part", columnList = "cfr_title,part_number")
})
public class Regulation {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Column(name = "cfr_title", nullable = false)
    private Integer cfrTitle;

    @NotNull
    @Size(max = 50)
    @Column(name = "part_number", nullable = false, length = 50)
    private String partNumber;

    @NotNull
    @Size(max = 1000)
    @Column(name = "title", nullable = false, length = 1000)
    private String title;

    @Column(name = "content", columnDefinition = "TEXT")
    private String content;

    @Size(max = 255)
    @Column(name = "agency_name", length = 255)
    private String agencyName;

    @Column(name = "word_count")
```



Technical Debt: Grabbing Regulatory Content

Testing was slowed down as I was putting in these API calls in Postman only to be returned with anti-web scraping messages.





Seems the instructions on the api from ecfr.gov were vague and unclear. The proper url request needed to be something along the lines of https://ecfr.federalregister.gov/api/, and even after making that discovery, it was unclear how I would populate my database with proper content so I could run a WordCount service over them.

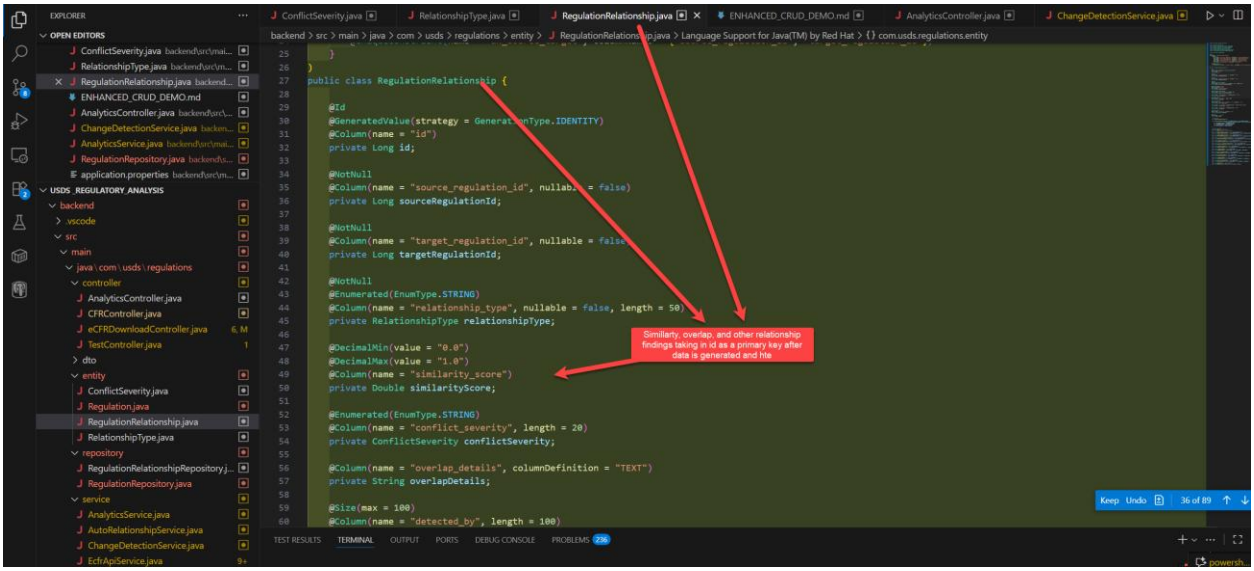
Ultimately, I decided to mock this missing data under MockDataService. For now, just invoke *POST http://localhost:8081/api/generate-mock-data-all-titles-with-relationships/5* to create mock data as a placeholder wherein other services may assume that we not only pulled live metadata correctly, but also content such as what you'd find here. \

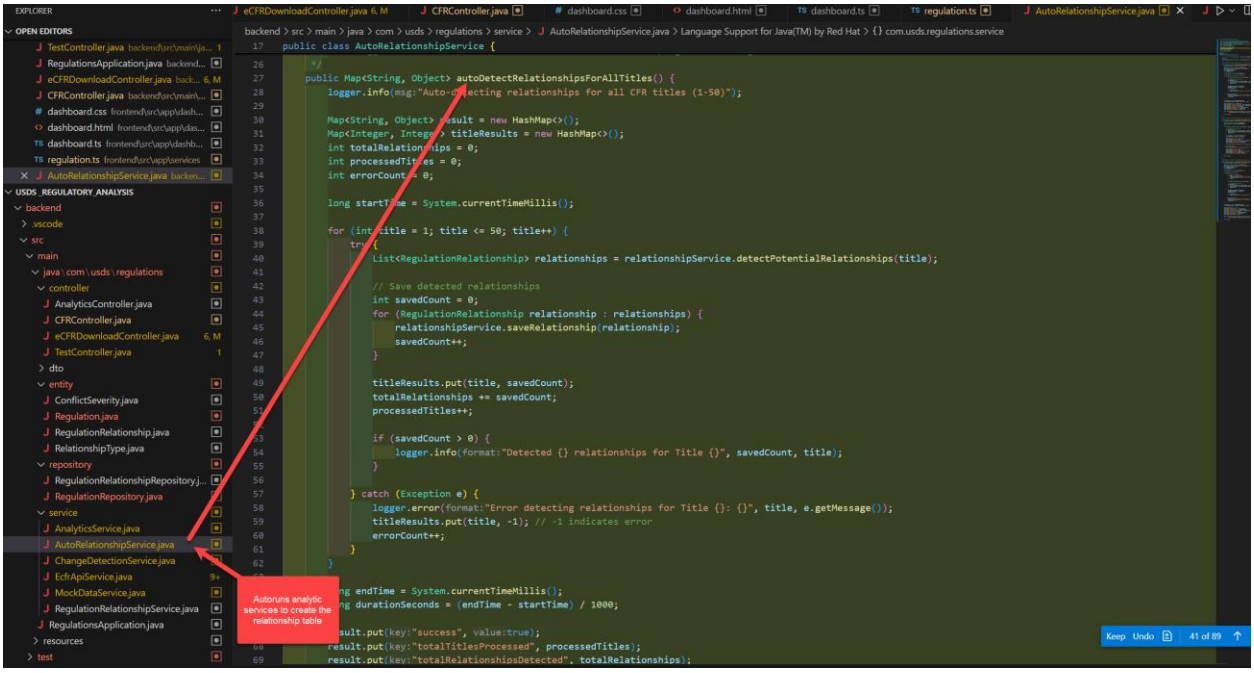
Further Metric Considerations

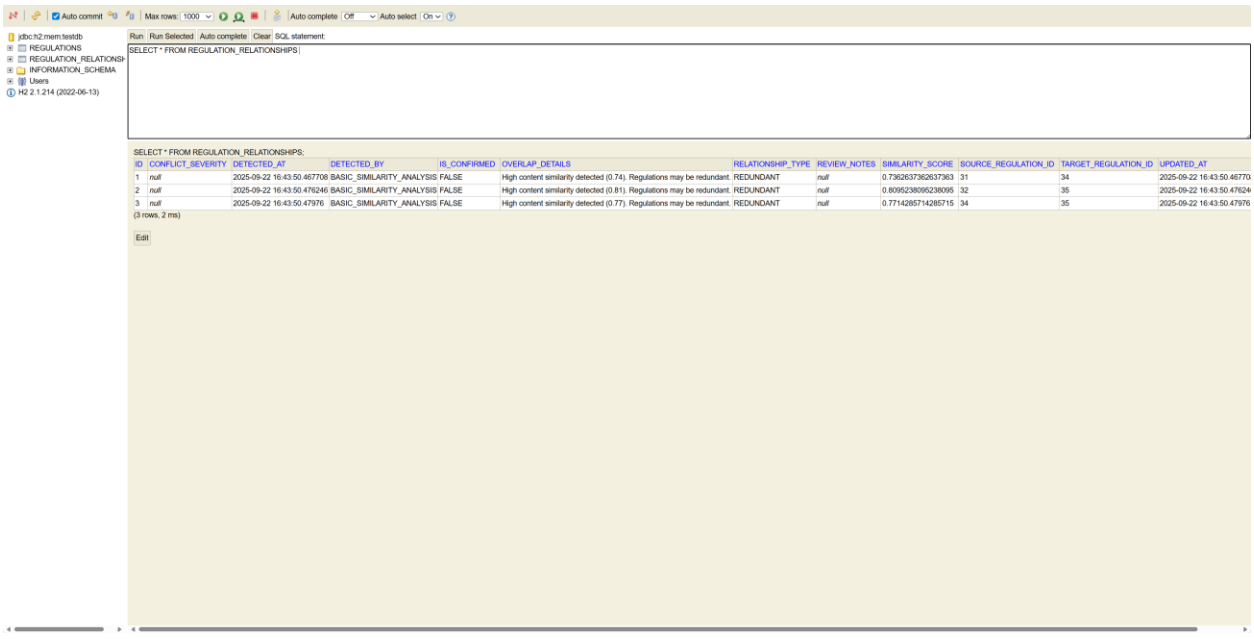
The assignment calls for **word count**, **historical changes over time**, and a **checksum** per agency but also asks that I consider metrics of my own.

Personally, I've always been interested in redundancy ever since I discovered how many regulations might be hitting the same thing years ago, so I ensured my mock data had the potential for overlap before forming a *RegulationRelationship* table that'd do some quick analysis on where we might have such redundancy, with differing scores. So within my mock data my goal was to create fake regulations with differing fake relationships.

A long term new feature would be AI generated sentiment analysis to review and check for redundancy in the form of a database agent, but such would require API keys and what have you, so it's simply rule based for now, with different assessment levels. This would be a full scoped feature involving a database agent and AI Management Architecture (such as here) that checks for content and looks for titles that overlap with each other.


RegulationRelationship.java
backend > src > main > java > com > usds > regulations > entity > RegulationRelationship.java > Language Support for Java(TM) by Red Hat > {} com.usds.regulations.entity
Similarity, overlap, and other relationship findings taking in id as a primary key after data is generated and hte


AutoRelationshipService.java
backend > src > main > java > com > usds > regulations > service > AutoRelationshipService.java > Language Support for Java(TM) by Red Hat > {} com.usds.regulations.service
Autoruns analytic services to create the relationship table

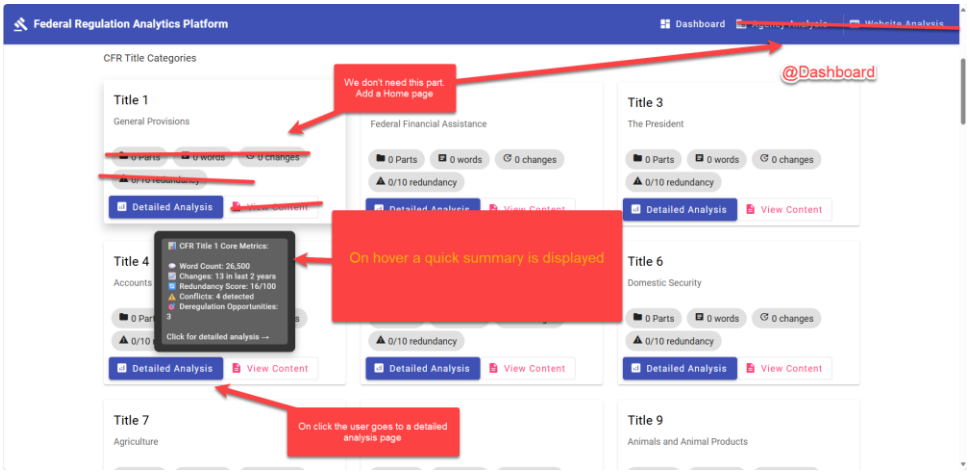


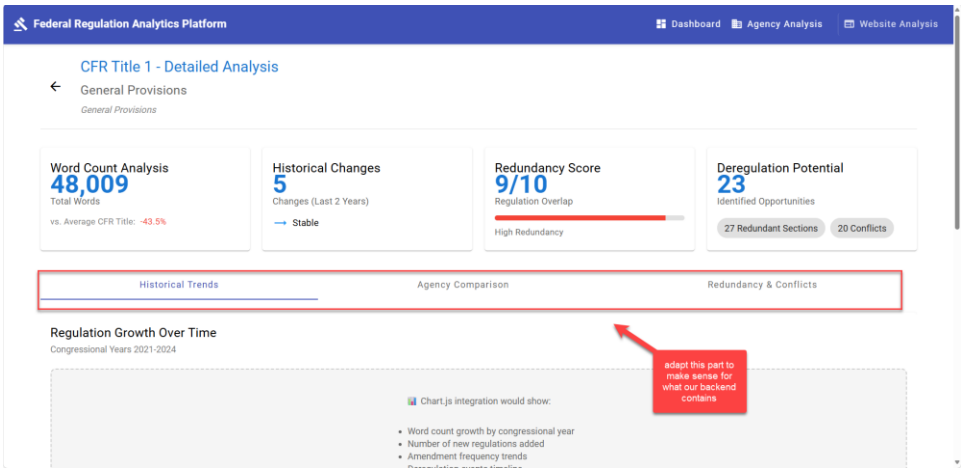
Frontend

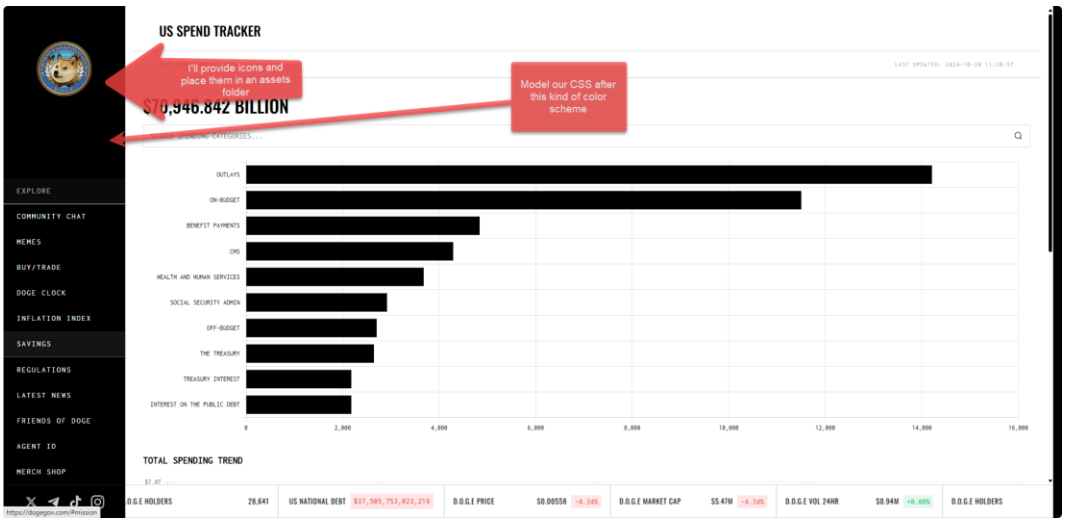
This part started around 4:30 pm roughly. It started by first looking at what DOGE websites already use for a color scheme and doing my best to imitate it.

The frontend was more straightforward. Much of it I was able to allow Claude to gin up because I chiefly dedicated my *human effort* into creating visual guides, mock ups, and pseudo whiteboards for it (Take for instance a few SNAGIT examples I wanted to still retain from the warmup activity from Friday).

Sample Training Mockups

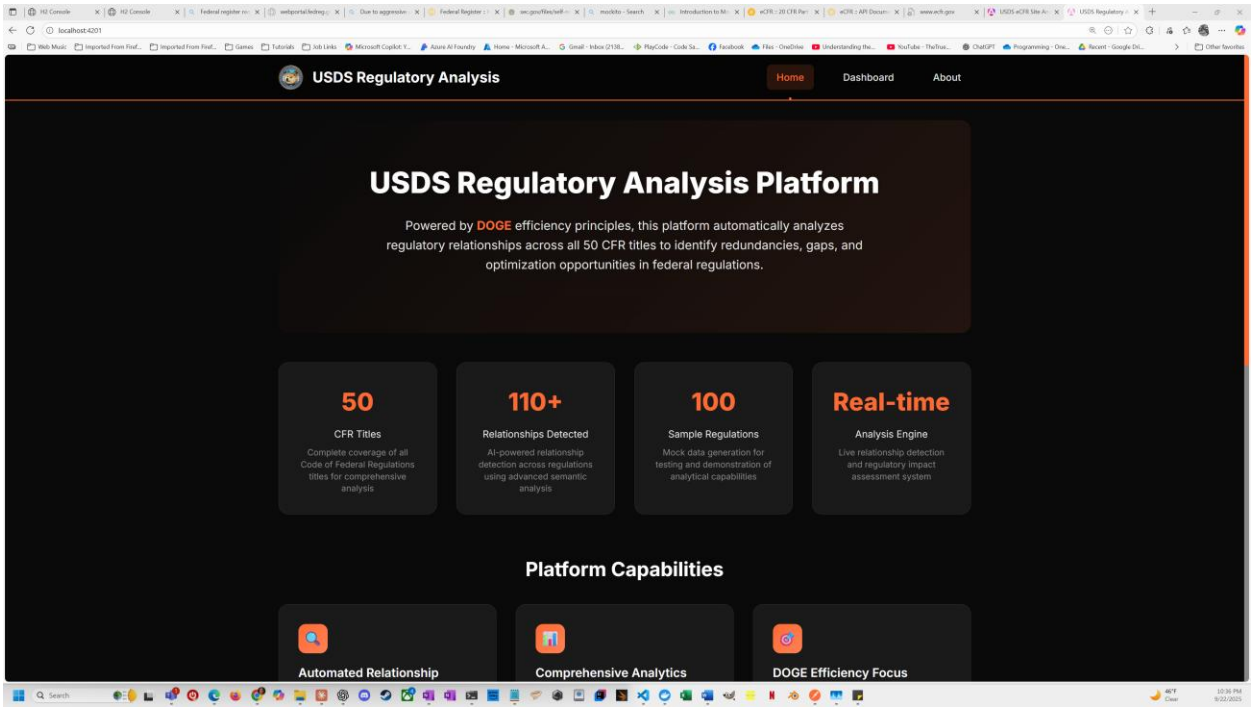


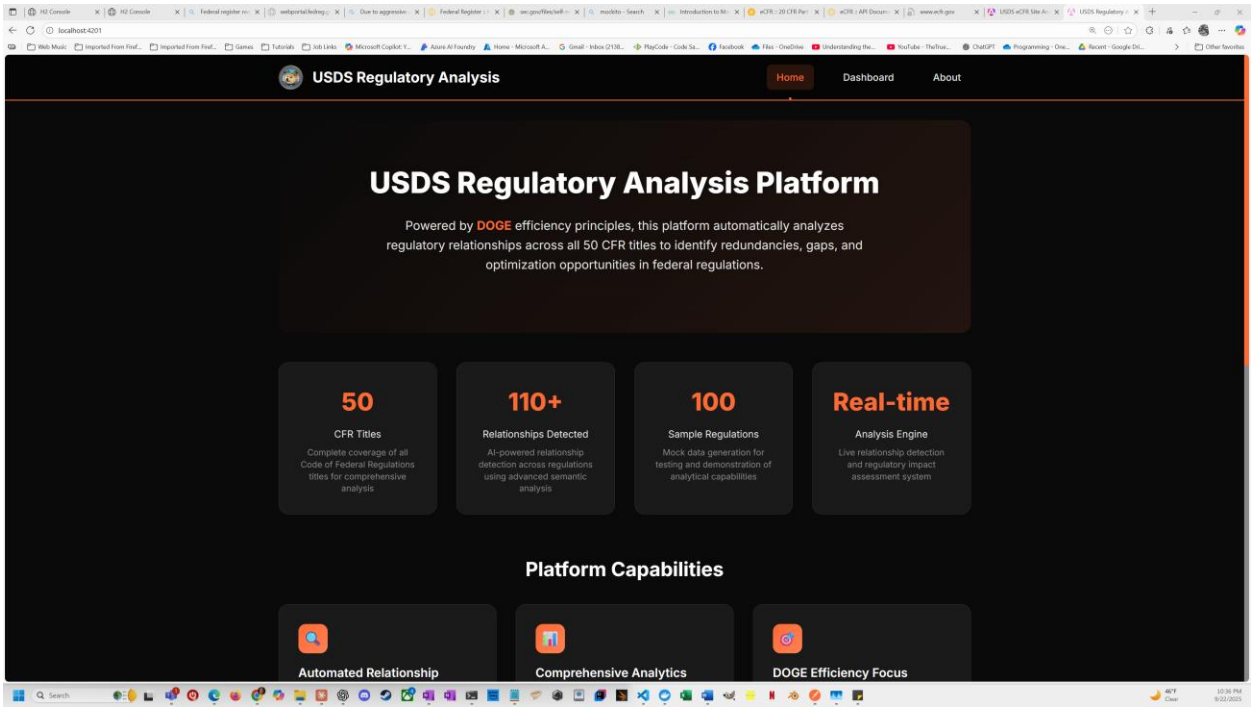


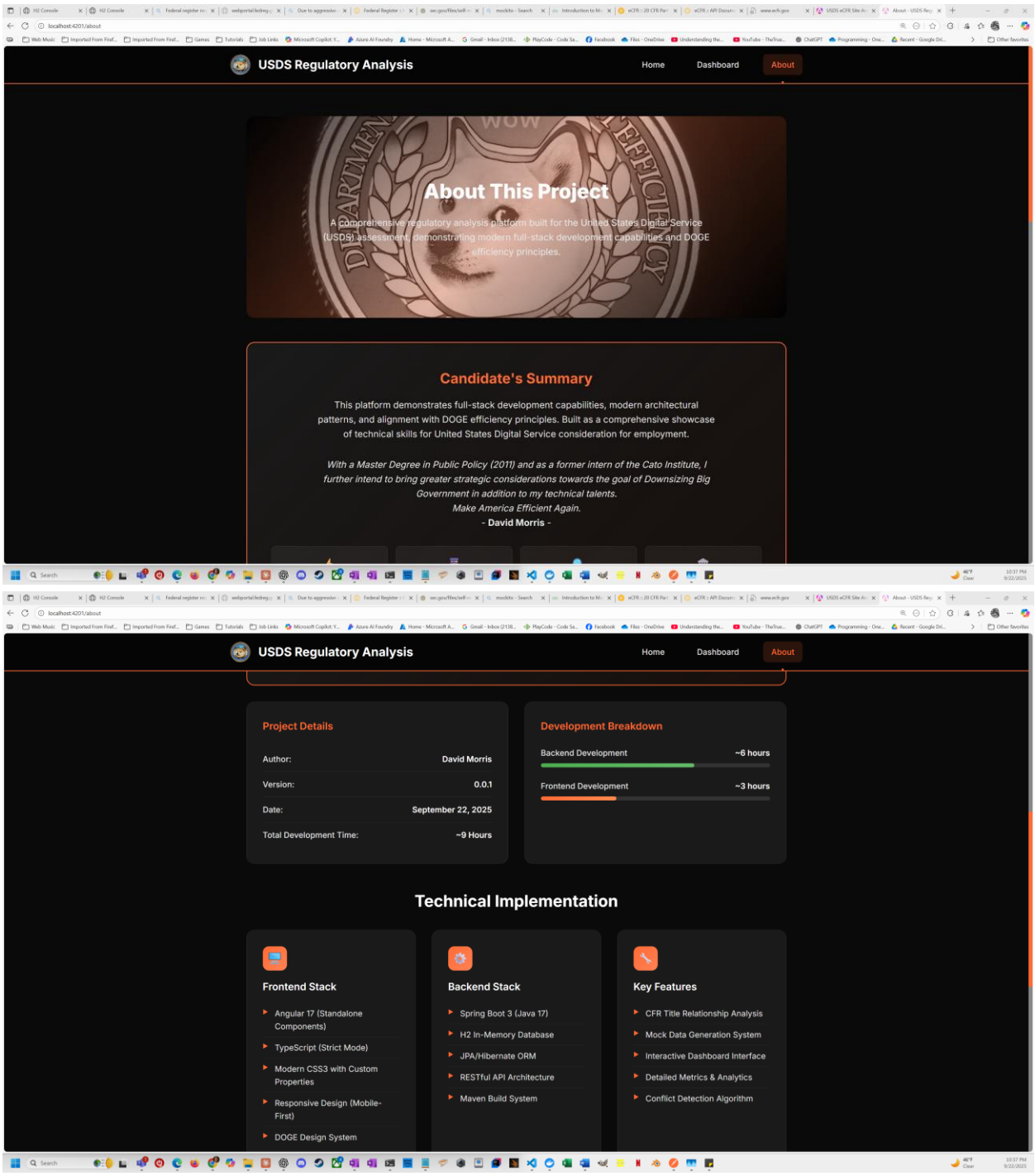


By being visual training-data oriented, I could afford to spend more time making sure the minimum viable product looks good and professional.

Final User Interface Screenshots







With time to spare, I put a considerable amount of *human effort* into the About page you'll find. Please check it out, for it contains a personal final message from me, as well as the time summary on this effort.

Roughly done by 7:30 pm. Then about an hour was spent writing this rough documentation. Naturally you'll see some extra cleaned up technical documentation in the longer run, but hopefully this, along with the READMEs and the website itself, will suffice.

Conclusion

I appreciate USDS for considering me as a candidate to join the team as a developer. I look forward to contributing to improved efficiency. Let's Make America Efficient Again!

Sincerely,

-David Noble Morris