Computer Science 2 Honors Research Report

Knowledge Base Web Application

Michael Paterson

Las Positas College

CS 2: Computing Fundamentals II

Professor Carlos Moreno

December 10, 2023

Introduction

For those who read a lot, watch documentaries, listen to podcasts, or perform research of any kind from a variety of media, it can be a challenge to centralize and connect ideas from disparate sources into a common place. A pile of notebooks, shoe boxes full of index cards, and messy word documents are often sought to mend this problem. An engineer can look at a problem such as this and formulate a solution. Through this project I set out to do just that: develop a software based common place to store knowledge. Fittingly, and exceedingly creatively, this project was named the Knowledge Base App.

This report reflects on the stages of design and implementation, the technologies used, challenges surmounted, and wisdom attained from building the application.

I. Pre-Design: Exploring the Problem Domain

The journey of this project began with exploring what software architects would refer to as the domain of the problem. What is it that we are solving for? The better of an idea one has of the problem, the more effective their solution will likely be. It is one thing to say I want an application that helps me stockpile information I find when scouring books, articles, and documentaries, it is another to define what it means to be a book, an article, a documentary, or anything, in terms of a software solution.

How do I go about saying “I want to take note of a piece of information” in a terminology that can make use of a computer’s hardware? Or in more lofty, abstract terms: run on Google Chrome? This is the goal of researching the problem domain. I needed to find the terms, subjects, adjectives, nouns, verbs, that define the essence of the problem.

In the industry this problem domain exploration is often done through meetings between designers, engineers, and the clients hiring them to solve a problem. A series of interviews is performed so the engineers know exactly what, for example, the record of sale of a bicycle details for a sporting goods store. A manager or accountant may tell the designers that every sale has a list of the products sold, their prices, the sum of those prices and a total calculated after applying a sales tax. The engineers can say, “ok, every product has a price, and on every record prices are summed and a tax is applied, product, price, tax. Great, now what does a product have? Surely it has something like a name, color, weight, etc. Tell me more about products, what is a product composed of?” This conversation goes on to define nouns and verbs, adjectives or qualities and descriptors of every part of the business they are helping. A physical receipt or daily sales report may even be handed to the engineers to look at to find information. This

process is how engineers learn the domain of the problem so that they can develop the domain of the solution.

I set out doing this for my own project. I interviewed myself by performing the regular task of reading a book, finding something interesting, and storing that information the way I normally would. I pulled out an index card, filled it out with the title of the book, author's name, page number and the excerpt of text I wanted to store. Alternatively I would take a notebook, write the excerpt and then reflect on it in my own words.

I began to see where my problem domain lies: I am storing this information on pieces of paper. These pieces of paper can be lost or destroyed easily, and are hard to access unless I take them with me. If I instead could store this information on a computer, and even better access it from the web so a number of devices could access it, then I could improve this system in a major way. In engineering terms then I must migrate a paper-based database to a web-accessible software database. My mission is to create a web-accessible application that allows users to store, access, manipulate, and destroy their own excerpts from source materials, citations describing the source materials, and freeform notes.

II. Design: Mapping the Solution Domain

Now that I had a mission statement, a clear idea of what I am solving, I could approach a design process. I interviewed my father, an experienced software architect familiar with web development, to get an idea of how I turn my idea into a functioning program. According to his advice, I needed a tiered architecture. One tier for managing data and for performing logical tasks on that data, or the actual processes that make my idea function, another tier for handling

access and messages to that data tier, also called a programming interface, and a tier that users can see and interact with to actually use the program the way it is intended to be used.

This is a three-tiered architecture: data, application programming interface (API), and user interface. I created a diagram that reflects this architecture:

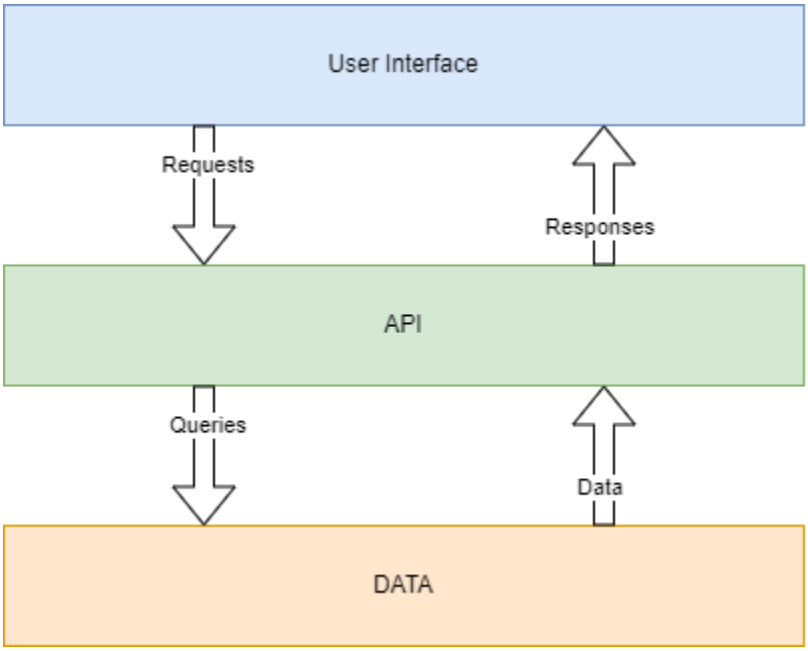

Figure 1. *Application Tier Architecture. Requests are made from the User Interface, handled by the API, and if valid, are used to query the Data tier, which is packaged and sent back to the User Interface as a response.*

Each tier of this architecture is constructed differently and operates on different programming principles, so I had to find the technologies needed to fill each of these roles.

For the data tier, a database management system was needed. There are two popular options for a database: relational or non-relational. Relational databases were coined by E.F. Codd, an IBM researcher, and work by associating information from different tables, or relations, in a database. Data is structured following a set of rules and restrictions that maintain referential, relational integrity through the database. Non-relational databases rely on different structuring

techniques, such as document stores, whose fields and values are strings of text that can be formatted in a number of ways that are later parsed by the programs that make use of the data. For the purposes of my application, and for a technical challenge, I chose to use a relational database for the application.

As for the middle tier, or API, I needed a robust general purpose programming language that can perform the majority of the tasks in the application. Popular choices in this category are C++, Java, C#, and Javascript. I am familiar with all of these languages at varying levels, but chose C#, my least familiar language per recommendation of my father, as the libraries and tools that C# and the .Net Framework it runs on are richly featured, and would likely handle the requirements of my application.

Finally, the user interface will run in a web browser, and therefore must use the languages of HTML, CSS, and Javascript. There are a number of frameworks that make working with these frontend technologies more efficient, and the frontend ecosystem is rapidly evolving. Amid the myriad choices for a frontend technology, I chose Angular, a framework for creating user interface components that contain both an HTML template for content, and Typescript, a version of Javascript that adds features for type-safety such as type declarations and errors, compile-time warnings and errors, and enums and interfaces for programming consistently.

With these technologies chosen, I could now dive more deeply into the lower-level design and implementation phases of each tier. It was time to begin creating the application from the ground up.

III. The Foundation: Creating a Database

An application is largely structured on its data. The information passed through the parameters of functions, stored, accessed and mutated, is what makes programs useful. It is essential then that a well thought out, sturdy foundation is laid for this project in the form of a properly designed database.

For guidance in understanding and designing a relational database, I read the book, "Database Design for Mere Mortals" by Michael J. Hernandez. In it, Hernandez describes the process of designing a database as such:

1. Define a mission statement and mission objectives.
2. Analyze the current database if there is one.
3. Create the data structures.
4. Determine and establish relationships
5. Determine and define business rules.
6. Determine and establish views
7. Review data integrity.

Following this process, I recognized that there are mostly subjects directly related to the purpose of the app, such as source materials (e.g. books), excerpts from those books, citations about the book itself, and freeform notes. These are all subjects of their own, with their own attributes, and as a result should become their own tables in a database. In addition, there are users who use this application and enter most of its data. Therefore a user profile must be its own subject with its own attributes. All of these entities are related in one way or another according to

the rules and theory of relational databases. To describe these entities and their relationships, I created a Entity Relationship Diagram:

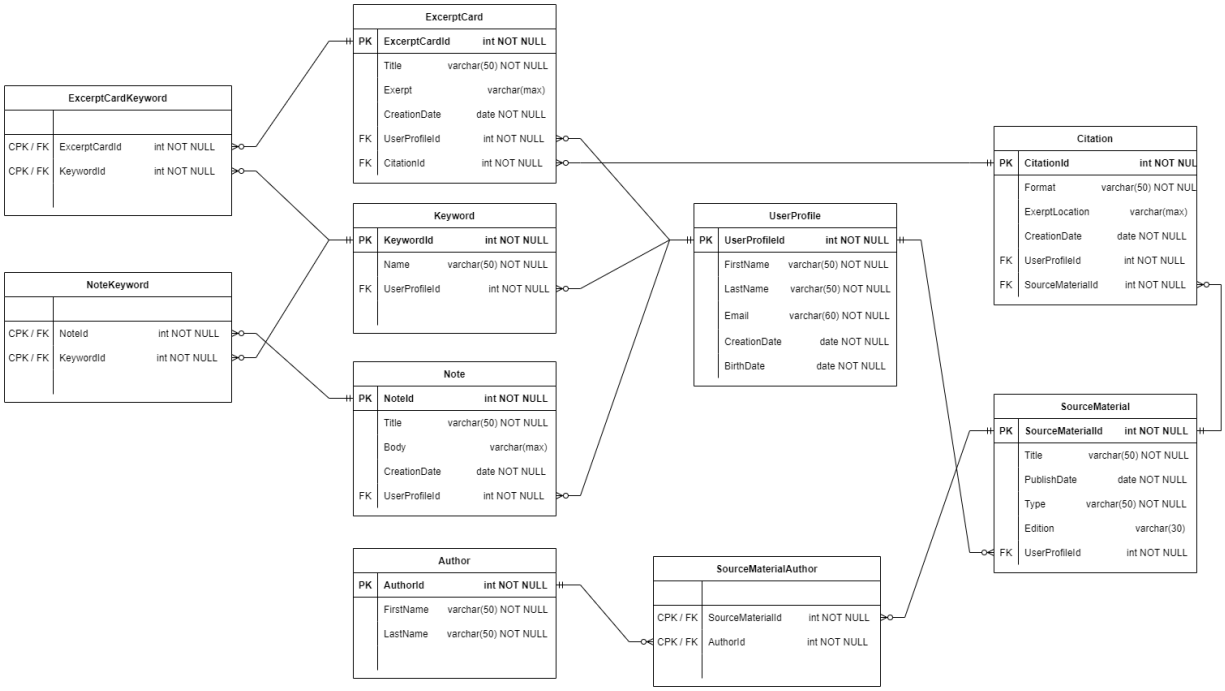

Figure 2. *Entity Relationship Diagram (ERD): Each table shows an entity, the rows in each table is an attribute. Following the rules of relational databases, each record of a table has a unique primary key, its ID, and any attributes from related entities are represented by foreign keys. These relationships are drawn using lines, whose markings indicate the type and participation level of a relationship. Ex: one UserProfile has many Notes.*

With each entity of my logical database mapped out, I was ready to implement it into a physical database. Relational databases are typically worked with using Structured Query Language, or SQL. I learned SQL from the Pluralsight course: “Introduction to SQL” by attorney-turned-engineer Jon Flanders. By using DDL, or Data Definition Language, a subset of commands in SQL, to implement the database and its tables, I was able to get a database

powered by SQL Server up and running. My next step was to create a program that interacts with the data in this database.

IV. The Structure: Developing Ways to Use the Data

In order to use my SQL database, I had to use a different subset of SQL called DML, or Data Manipulation Language. DML is how I can add, retrieve, update and delete data from the database. While I could hand-write commands in SQL every time I needed to interact with the database, it would become an arduous task to maintain that level of work. It would also be very limiting if I couldn't perform complex logical tasks on data, such as checking if an email and password combination actually exists in the database.

I therefore needed to step into the general purpose programming technology I chose to create a way of translating the relational data into something that can be worked with by more complex programs. To get SQL data in a format that can be used in a C# program, I needed to employ Object-Relational-Mapping. Luckily .NET has a tool called Entity Framework that does just that, by making SQL queries from C# function calls, retrieving the information from the SQL database, and using that to provide context to my C# programs. Any work that needs to be done to the data, such as saving, modifying, removing, or calculation is done in this layer of the program with the database context provided by Entity Framework.

Entity Framework is perfect for mapping data between my middle tier and data tier, and is incredibly powerful for both data manipulation and data definition. It is essential, then, that I keep a handle on the reigns of this tool, and only allow it to be used in just the way I intended, to create, read, update, and delete specific records in the database, and preventing unwanted

behavior such as requests to create new tables, or sending queries that delete all the data in the database. A dangerous possibility indeed!

To solve this vulnerability, it was important to encapsulate the functionality of my data-access and data manipulation layer, what I called the “domain” layer, from the outside world, and expose only approved ways of interacting with the database. For each entity (also referred to as model) in the domain I developed an interface, essentially a contract between the implementation of an entity and anything that is a client, or user, of this entity. As long as the client promises to use the exposed functionality with valid parameters, the implementation promises to return a valid result.

This same idea is used on an application-wide scale, to protect my domain layer and database from non-users, unapproved clients to the domain, and potential threats from the outside world. This level of protection and regulation is called the Application Programming Interface, or API.

V. The Walls: Building an API

The Application Programming Interface, or API, is the controlling mechanism of the application. It ensures that only valid requests from approved sources can make it through to the database, assembles the responses into data transfer objects, and sends them back towards the requester. This API has no state of its own, uses HyperText Transfer Protocol (HTTP) as a medium for communication, a uniform structure, and caches data through the domain layer to the database. Therefore this type of API is called a Representational State Transfer API, or REST API. A REST API will also typically send the data it retrieves from the domain by mapping it to

a data transfer object, often formed in Javascript Object Notation or JSON, a format easily accessible by frontend programs written in Javascript or Typescript.

A REST API is easy to work with for both the client side and service (domain) layer because of its uniformity and streamlined architecture. In order to implement a REST API for my application, I employed ASP.NET, the tool provided by Microsoft for just this purpose, to expose controlled endpoints of the application's URL address to the outside world. Any requests sent to the endpoints are handled through their respective controllers and validated. If valid, the data in the request is worked with in the domain layer.

In addition to providing controlled endpoints, the API handles authenticating the users sending the requests, and authorizing them to make those requests to the domain layer. With that, the protective, controlling layer of the application is complete. The final tier of the application is a way to actually view the data and functionality of the application. It could prove difficult if someone with no programming background had to talk to an application programming interface, instead it is better to have an interface just for end users, aptly named the user interface.

VI. The Doorway: Making a User Interface

The user interface of the application is the portal through which end users may interact with the program. It is the front end of the application, and the client of the backend server containing the API, domain layer and database.

To design how the user interface should look, I used a program called Figma. It is an image editing app specialized for designing user interfaces and creating mockups to test the user experience of the application. And although my user interface design skills are crude, I was able to make a view for the application that wasn't a complete eyesore, the colors were a nice dark

greyscale pierced with a purple accent. The home page featured a watercolor portrait of the great thinker Marcus Aurelius, booming with splatters of water color that brought much needed energy to the home page.

In addition to the look and feel of the user interface, it also needed to provide the means of entering and retrieving data to and from the backend. This is done through the use of forms. My father would humorously mantra, “the internet is forms and data”, and he couldn’t be more correct. To create a new user profile, log in, write and save notes, excerpts, and citations, the front end program needed to present forms to the user to be filled out, ensure their input is valid according to the rules of the application, and then call upon services to communicate this input to the backend via the API.

After a cycle of creating HTML templates to provide structure to the website’s components, CSS to style that structure, and Typescript to give each component the plumbing it needed to circulate data for every feature of the application, the front end was complete and functional. The first requests, trickling down from the user interface to the database, and pages hydrating with “200 OK” success responses from the API meant that the core features of this application were fully functional. My work here was done. Well, sort of.

VII. Conclusion

The core features of my application are complete. I am greeted by an inspiring watercolor portrait of Marcus Aurelius, logged in, and able to create notes and store excerpts to my heart’s content. However, there are a plethora of features I am now inspired to add to this application. I want to finish the keyword functionality that would link different notes and excerpts together, along with a search engine to power that functionality. I want to explore implementing a graph

database and parse every entry for a user to create more fluid connections of ideas. I want users to be able to explore each other's work and collaborate. I want the site to look pretty and be bug free.

Even though the project is ready for its first deadline, December twelfth, 2023, it does not spell the end of this journey. It is a proof of concept and invitation to continue to experiment with features and solve more problems. It is the baseline of a song waiting for its melody. I'm looking forward to what my growing mastery of these tools and engineering prowess can muster in the future, and where this work will take me. I'm ready to get to work.

Note

If you would like to see the code for my application, it is available to view here on GitHub. At some point the application will also be hosted as a website for demonstration and public use.

Thank you to everyone involved in the creation of this application, from the industry knowledge of my father to the scientific insight, curiosity, and supportiveness of my professor Carlos Moreno. Without their mentorship this project would have been a dream in name only.

References

Hernandez, Michael J. Database Design for Mere Mortals : A Hands-on Guide to Relational Database Design. Upper Saddle River (N.J.), Addison-Wesley, Cop, 2013.

“Online Courses, Learning Paths, and Certifications - Pluralsight.” Www.pluralsight.com, pluralsight.com/. Accessed 11 Dec. 2023.

BillWagner. “.NET Documentation.” Learn.microsoft.com, learn.microsoft.com/en-us/dotnet/.