THE DESIGN AND IMPLEMENTATION OF A TEACHING WIKI

A Thesis
by
WILLOW EMMELIINE SAPPHIRE

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

May 2022

Department of Computer Science

THE DESIGN AND IMPLEMENTATION OF A TEACHING WIKI

A Thesis
by
WILLOW EMMELIINE SAPPHIRE
May 2022

APPROVED BY:

____
Cindy Norris, Ph.D.
Chairperson, Thesis Committee

____
James B. Fenwick, Jr., Ph.D.
Member, Thesis Committee

____
Alice McRae, Ph.D.
Member, Thesis Committee

____
Rahman Tashakkori, Ph.D.
Chairperson, Department of Computer Science

____
Marie Hoepfl, Ed.D.
Interim Dean, Cratis D. Williams School of Graduate Studies

Copyright © 2022 by Willow Emmeliine Sapphire
All Rights Reserved

Abstract

THE DESIGN AND IMPLEMENTATION OF A TEACHING WIKI

Willow Emmeliine Sapphire
B.S., Appalachian State University
M.S., Appalachian State University
Chairperson: Cindy Norris, Ph.D.

Web-based courses are a mainstay of modern learning. There are many websites that offer tutorials and courses on nearly every subject. These courses are made by either individuals or organizations. This thesis develops a website, Wiki Courses, that applies the wiki model to online courses. Users of the website are able to create and edit any course they wish. The content of every course is crowd-sourced by people on the internet and not controlled by any individual or organization.

Wiki Courses uses Git as a database to store edits made to each course along with its content. Saving this extra information provides editors the ability to undo changes. Using Git as a database removes the need to manually create a system to store edits as Git already saves this information. This significantly speeds up the process of creating a wiki. Git is not traditionally used as a content database and so using it as such brings a number of challenges. This thesis explores the effectiveness of Git as a database.

Acknowledgements

I am unable to adequately express how grateful I am to those who helped create this thesis. I did not do this alone and am incredibly thankful to everyone who supported me. First I want to thank my thesis advisor, Dr. Cindy Norris. As I developed my thesis from the initial idea to the completed work, she provided invaluable ideas, suggestions, and advice. Many of the main concepts in this thesis started out as ideas from Dr. Norris. This thesis would not have been the same without her.

Second, I want to thank my parents who have been a constant support throughout my time in college. While writing this thesis, they provided a helpful ear on which to bounce off ideas. Their help and support have been critical to my success, not only in this thesis, but in my academic career as a whole.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

A wiki is a type of website that crowd-sources information by allowing many people to edit the site content. Wikis are made up of a series of interconnected web pages that often contain many links to both internal and external resources [10]. In most cases, wikis act as databases of knowledge, providing detailed information on a set of topics [8]. This thesis explores the creation of a wiki in which each page is a course aimed to teach the user about a subject. This website is titled Wiki Courses. The use of a wiki for course creation allows multiple, possibly unrelated instructors, to simultaneously develop the course. Wikis store the revisions and edits made to them in addition to the content [10]. This allows edits to be undone if needed. A unique contribution of Wiki Courses is that it uses Git, a distributed version control system, to store the content and information about edits.

1.1 The Power of Wikis

Wiki comes from the Hawaiian term *wiki wiki* meaning quick. They were given this name because they can expand very rapidly since any user is able to edit and add to them [12]. The ability to grow quickly is one of the main advantages of wikis. When an organization or individual is solely responsible for website content, that content will take much longer to be created [12].

One of the best examples of a wiki is Wikipedia. Wikipedia has created a huge database of knowledge through crowd-sourcing and is one of the most trafficked websites in the world [10]. Another excellent example is the website Fandom. Fandom itself is not a wiki, but rather a platform for users to create wikis. Many games, TV shows, books, and other media have wikis created on this platform. These wikis are often the primary resources for people who consume the related media. In the case of these wikis, crowd-sourcing is not just important to create the database quickly, it is also the only source of the information. Many games and TV shows do not have a governing body that would be able to create such a website. Therefore the users must create the database themselves in order to have it. This makes wikis a valuable tool for these users and enables them to share information in a way they otherwise would not be able to.

Another benefit of wikis is that they combine knowledge from many people, which can help reduce biases and ensure all viewpoints are included. In addition, when many people are working on the content, errors are more likely to be detected and the most necessary content is likely included. Wikis allow any user who sees a flaw to fix it, which improves the overall quality of the content.

1.2 Git as a Database

Since wikis are edited by the general public, there is always the danger that a user may maliciously change a page for the worse by deleting sections or making substantial inappropriate edits. Most wikis have a number of methods to help combat this problem. One of the most important methods is the ability to revert a page to a previous version. This allows users to undo bad edits without the need to manually rewrite the page. Performing these reversions requires some sort of version control management.

Git is a version control system typically used to store code for programs being developed [18]. Git allows users to revert their code to older versions and track all the changes made. If a change breaks the program, it is possible to revert the code back to the last working version. It is important to note that standard use of Git is for development, not production. End users of the product have no interaction with Git. This is not the case for Wiki Courses.

Wiki Courses attempts to leverage the version control system of Git by using it as a database. This means that end users unknowingly interact with Git. When a course is requested by the user, the server retrieves the course information from the Git database and returns it to the user. This is a novel usage of Git and it comes with a number of pros and cons. Wiki Courses explores how Git can be used in this manner and the benefits of doing so.

1.3 Introduction to Wiki Courses

Wiki Courses is a wiki website where every subject is a course. There are no restrictions on what the courses could be about. The main page of the website (Figure 1.1) has a list of courses displayed, along with a search bar allowing users to search by course title. The list of courses automatically updates as the search term is entered. The user can then select a course and it will expand to display its description. Beside the description there is a button that the user can click to go to that course page. Below the list of topics is a tool to allow the user to determine how many topics to display on one page and a button to go to the next or previous page of topics.

Each course is made up of a list of topics. When a user navigates to a course they are taken to the first topic, which is always the course description (Figure 1.2). There is a navigation menu to the left of the course page that can be used to go to any topic. There are also navigation arrows at the bottom of the page to move to the next topic or previous topic. Currently, topics are only text-based lectures using HTML formatting.

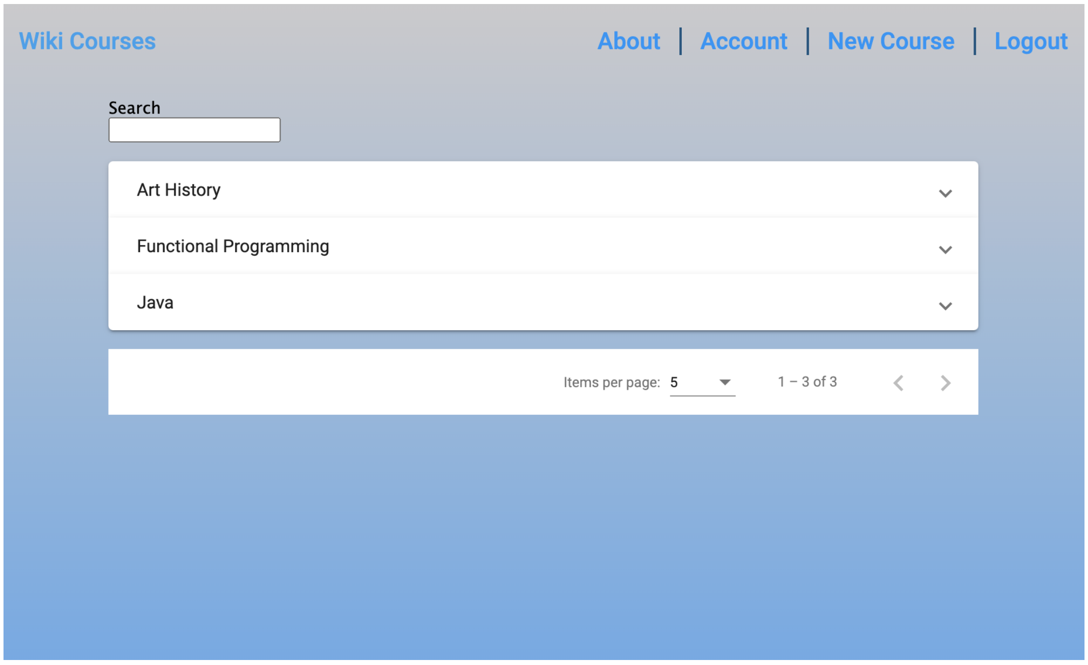
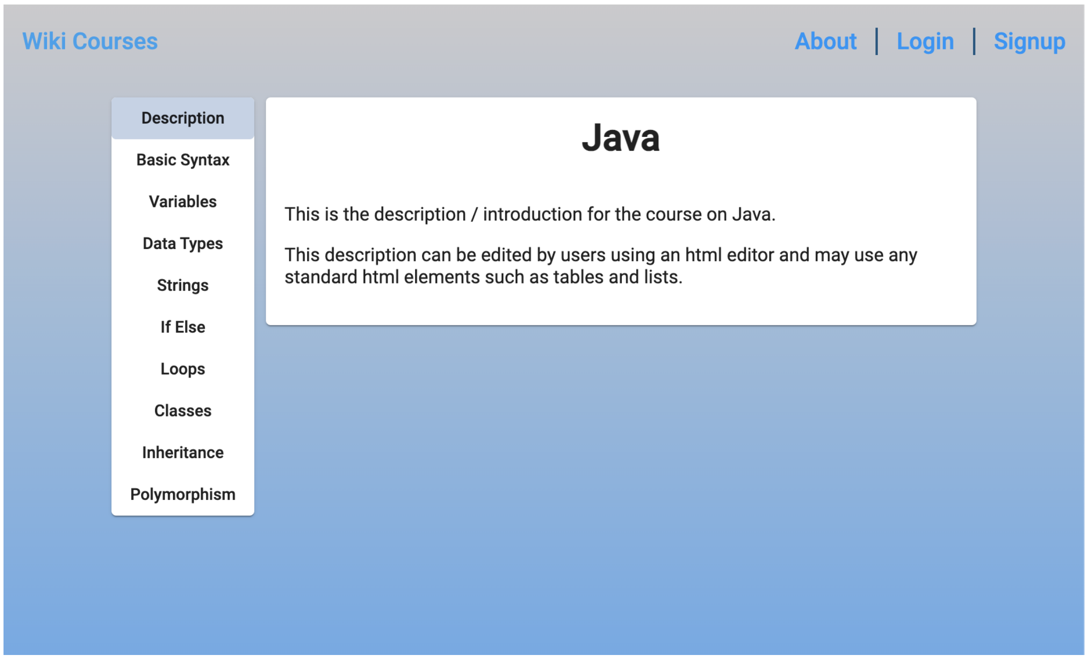


Figure 1.1: Wiki Courses Home Page



Java

This is the description / introduction for the course on Java.

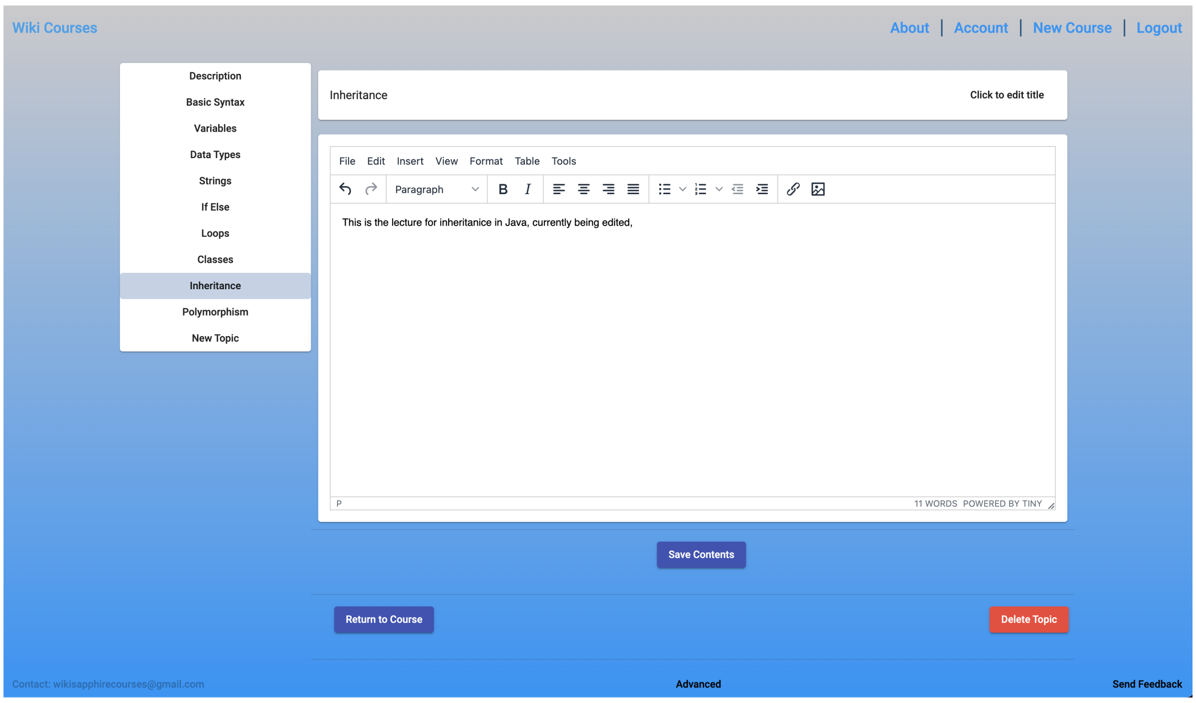
This description can be edited by users using an html editor and may use any standard html elements such as tables and lists.

Figure 1.2: Wiki Courses Course Page

The difference between courses and topics is that courses are much larger in scope than topics. A topic is a very specific subject while a course is an aggregation of topics that all relate to a larger subject. For example, a course could be *Java Programming*. Topics within that course might include classes, arrays, primitives, inheritance, and other knowledge relating to Java.

A core part of Wiki Courses is the ability to edit content. Editing course content is done through an html text editor (Figure 1.3). Courses can be created and edited by anyone but this cannot be done anonymously. Editing requires that the user be logged in to a verified user account. Wiki Courses has a user account system where users can create accounts and verify their emails. Email verification is done through a secure link that is sent to the user's email from the server. Clicking on this link will take the user to a page on the server that verifies their email and then redirects them to the website. Once logged in to a verified account, a button appears in the header to create a new course and a button appears on each course page to edit that course.

This thesis will discuss the usefulness of wikis and how they are implemented. It will discuss the design, implementation, and goal of Wiki Courses as well as the limitations in its creation. Finally, it will discuss how Wiki Courses was tested and how it can be improved upon in the future.



This is the lecture for inheritanice in Java, currently being edited,

Figure 1.3: Wiki Courses Edit Page

Chapter 2 – Related Work

2.1 Crowd-sourcing Education

Online learning, also called e-learning, has been a growing educational venue over the past two decades. Most e-learning materials follow one of two approaches: knowledge repositories or massive open online courses (MOOCs). Knowledge repositories contain information which learners can use to better understand a subject. However, the information is structured like an encyclopedia entry, not a course. These knowledge repositories are sometimes crowd-sourced and sometimes created by a governing body. MOOCs are structured courses that follow an institutional model similar to that of universities. Neither of these types of e-learning platforms provide courses created collaboratively by the public. However, there are some instances of e-learning platforms that leverage crowd-sourcing for their curricula.

2.1.1 Peer 2 Peer University

Peer 2 Peer University (P2PU) is an open-education online community. Any user of the platform can create their own course that other users can take. These courses are synchronous and have class sessions held by the instructor. The goal of P2PU is to create a collaborative, free, online learning environment [2].

Courses at P2PU cannot be taken at any time. The courses are held in a structured, scheduled format similar to traditional, in-person learning systems. Course materials are taught by the instructor. Signing up for a course as a participant indicates a commitment to taking the course and remaining active throughout. Courses involve active participation from the learners in the form of discussion groups, online forums, and group activities. Users may also register for courses as a “follower” rather than a participant. Followers receive updates from a course but are not committed to actively participating [2].

While the content on P2PU is crowd-sourced, the individual courses are not created collaboratively. When a user creates a course, other users are not able to edit it. Peer 2 Peer University is an example of creating a crowd-sourced collection of courses. However, each individual course is still created and managed by an individual [7]. Having elevated privileges for the course creator reduces the collaborative nature of the creation of academic resources.

2.1.2 Wikiversity

Wikiversity, a sister project to Wikipedia, attempts to crowd-source course materials and become a free online university. The courses offered by Wikiversity are free to take at any time and do not require signing up and do not have instructors [15]. The Wikiversity web pages appear very similar to those of Wikipedia. Each course contains a list of categories which may contain subcategories or pages. The pages contain information on the topic in the form of text lectures and images. Wikiversity also contains a number of other academic resources such as activities, flashcards, and articles. All of the content on Wikiversity is entirely crowd-sourced [26]. At the time of this thesis, Wikiversity has 59 primary subjects, each with a number of sub-categories. However, many of these categories contain no content and are page stubs.

2.2 Git Usage as a Database

Git is a version control system, or more specifically a source control management system, that developers use to store code [18]. Git tracks changes made to the code and provides a multitude of features for managing the stored code. However, Git is not a database management system (DBMS) and was not designed to be used as a content database. Git does not define documents with properties and relationships nor does it provide a query language. While Git was not designed as a DBMS, it does store data and provides version control on this data in such a way that it can be used in place of a DBMS in some situations. This is a relatively novel use of Git. While this is not the first project to use Git in this fashion, it is one of a small number of projects that are involved in exploring this space.

2.2.1 The LHCb

In 2018, the Large Hadron Collider beauty (LHCb) experiment team at CERN decided to use Git as a database [6]. CERN describes the LHCb experiment as an "experiment specializ[ing] in investigating the slight differences between matter and antimatter by studying a type of particle called the 'beauty quark', or 'b quark'" [5]. The LHCb relies on what they call a *conditions database* that stores information about the state of their equipment at varying points in time. LHCb originally used a database built on SQLlite, a standard relational database system. However, they found that there were limits to this system and decided to start using Git instead. Git was chosen because of its ubiquity, ease of use, compact file storage, and version control capabilities. The LHCb team built a front-end function library for their system to interact with the Git repository. They also stored information within the file system by defining the directory structure and file names [6].

The LHCb team found that their Git database performed better then their previous

SQLite database both in terms of access speed and storage size. The access speed was more than twice as fast as the SQLite database and the storage space was reduced by approximately 6 times. The team concluded that Git was an excellent database system, but noted that the system would likely not scale well [6]. The data stored by LHCb is ideal for a Git database since it has a clearly defined structure that is not subject to change or grow. If the schema changed or the amount of data grew significantly, Git would become difficult to use.

2.2.2 The Open Tree of Life

In 2015, researchers associated with the Open Tree of Life project developed an infrastructure for biologists to share phylogenetic data using Git as a backend database. They chose Git for it's version control system to track edit history, its familiarity to developers, and because they believed the editing pattern associated with their data would be well supported. They found that this system lowered the cost of entering data into the archive and was effective at supporting community curation. They also found Git helpful by allowing individual researchers to clone the database and maintain their own versions [17].

Chapter 3 – Background

Wiki Courses is built using a MEAN stack. MEAN is an acronym for MongoDB, Express, Angular, and Node.js. MongoDB is the database. Node.js and Express run the server and all server-side code. Angular is the client-side application. Node.js and Express serve a REST API for the Angular application to allow access to data on the server. Angular runs all client-side code as a single-page application (SPA).

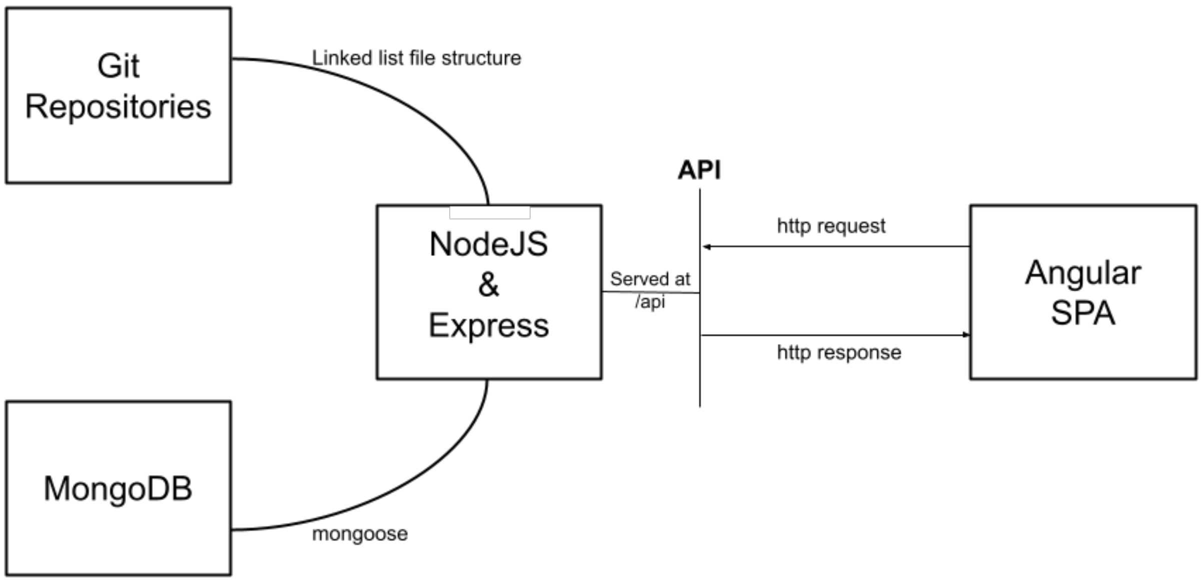


Figure 3.1: Wiki Courses Architecture

Figure 3.1 shows how all of these parts work together to create a functional website. Wiki Courses also uses Git as a repository for all its code. Both the Angular application and the

API are hosted using Amazon Web Services. Finally, Wiki Courses also uses Git as a part of its database. MongoDB is used to store user information and basic course information while the data for each course is stored in a Git repository created for that course. The remainder of this chapter provides an overview of these various technologies used to create Wiki Courses.

3.1 Angular

Angular is a JavaScript framework managed by Google and developed for creating single-page applications (SPA). It defines an organizational system for client-side code that promotes readability, re-usability, code composition, and separation of responsibilities. Angular employs a Model-View-Controller (MVC) based pattern, allows easy integration with RESTful APIs, and provides simple dynamic data binding, the cornerstone of a responsive website [1].

Angular provides the web page that users will see when they first come to the website. This page is a SPA that only contains one HTML page. The page has one large JavaScript bundle attached to it that updates portions of the page as needed [20]. The SPA allows the page to continually update without ever needing to refresh the browser. Browser refreshes can be slow and unattractive. Navigation between pages can be simulated by simply updating the content on the page with JavaScript so that the user still feels like they are interacting with multiple pages. Additionally, SPAs can be completely stand-alone, without any strong connection to a server. This makes RESTful APIs a perfect complement to them since they exist as a stand-alone back-end [20].

Two notable alternatives to Angular are ReactJS and VueJS. Both perform the same functions as Angular in web applications. There are many differences between the frameworks but they all accomplish the same tasks and therefore any could be used for any application. VueJS is a much smaller framework that does not define a file structure and can control just a small portion of a website or the entire website. It is ideal for smaller applications where Angular and ReactJS provide more functions than are required and add unnecessary are more attractive options. Both are good choices and there is not a strong reason to pick one over the other. Angular was chosen over ReactJS for Wiki Courses due to personal preference.

3.2 Node.js

Node.js is used as the server for the REST API from which the SPA can make requests. It is possible to host the SPA on the same server, but since they are completely separate applications they can also be on separate servers. The Angular SPA is hosted on a second Node.js server that has almost no logic and only serves the HTML page of the SPA. This HTML page is always returned regardless of the path, which allows the Angular application to handle navigation without interference from the server. A key benefit of Node.js over other servers, such as Apache, is that it is written in JavaScript or TypeScript. This allows the server-side and client-side code to be written in the same language, increasing code readability. The API allows for asynchronous data transfer between the database and the SPA, which is critical for maintaining a responsive web-page.

A RESTful API is a resource that can be accessed at a number of exposed endpoints. These endpoints allow applications to make HTTP requests for resources on the server. The API itself does not need to know anything about applications that use it, which provides separation of responsibilities. There is a set of principles that define a REST API.

- Everything is a resource.
- Each resource is identifiable by a unique resource identifier (URI).
- Requests use the standard HTTP methods.

- Resources can have multiple representations such as XML and JSON.
- There is no internal state [3].

Node.js employs an event loop to manage requests (Figure 3.2). An event loop works by sending operations to the database with a registered callback. The callback allows the loop to process other requests while the current request is being handled. When the operation is complete, the callback is triggered and the event loop returns the information to the requester. Additionally, the event loop only employs a single thread, which helps conserve system resources. Other similar technologies often create a new thread for each request, which can quickly consume resources and potentially cause performance issues [19]. One potential downside of this event loop is that if a normally asynchronous task, such as reading a file, is performed synchronously, all operations of the server must pause until that task is complete.

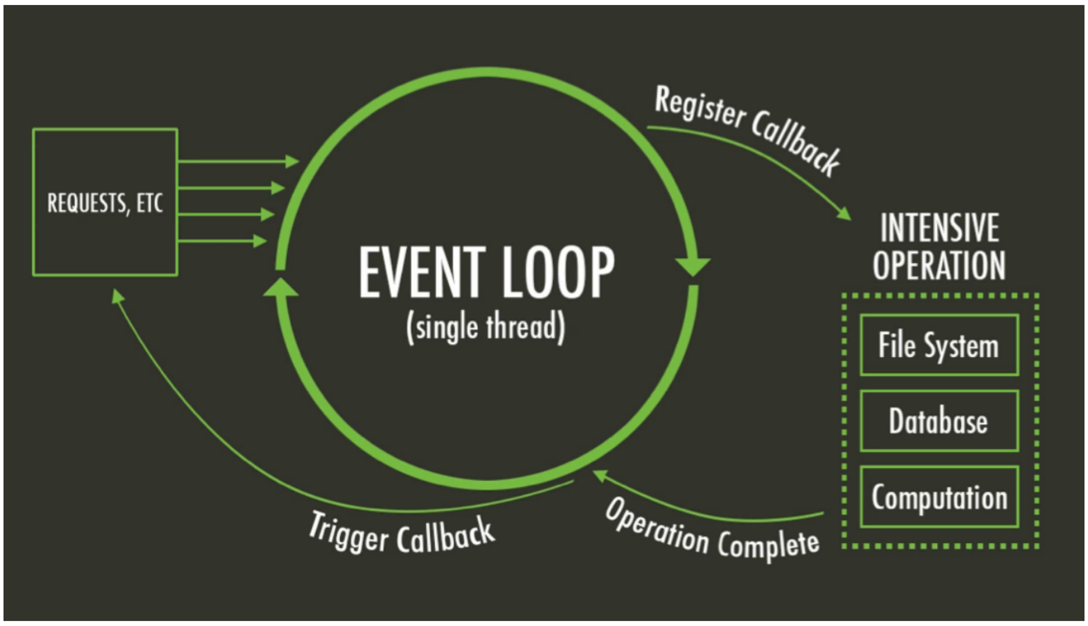


Figure 3.2: Node.js: JavaScript & The Event Loop. Reprinted from [19].

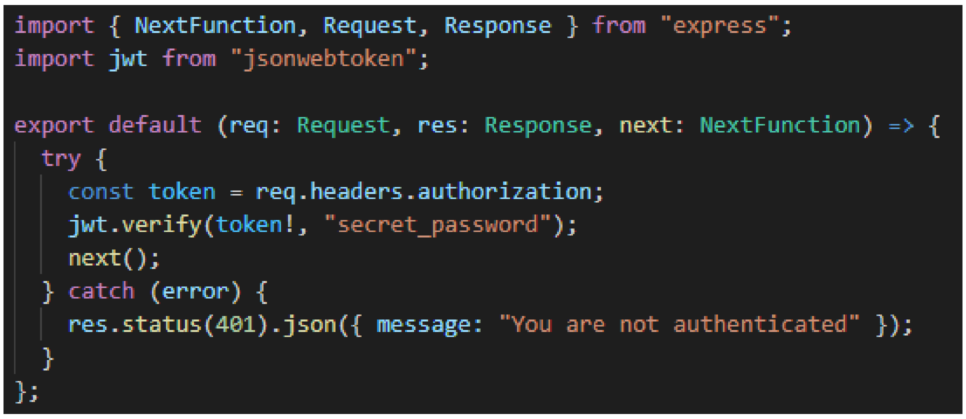
Node.js provides a package manager called npm (node package manager). This package manager provides a clean way to manage all the different packages used in an application. The package.json file, required by every Node.js application, includes a list of all package dependencies. These packages do not need to be stored in the project repository because the `npm install` command will simply read all the dependencies and install them. When using a MEAN stack, the packages relating to every layer of the stack are installed with npm, providing a very clean organization. Wiki Courses has two separate package.json files since it has two separate servers for the API and SPA. One file contains the back-end dependencies and one contains all front-end dependencies.

3.3 Express

Express is a Node.js framework that simplifies the server-side code. Express eases web application development with Node.js making it practically a ubiquitous partner [1]. Express is one of the most downloaded packages found in npm for this very reason [14]. Express makes the creation of APIs very simple by allowing the user to define *routes* based on the standard HTTP methods, create middlewares for those routes, and elegantly handle errors. Without Express, the development of a REST API for Wiki Courses would have been significantly more time consuming and resulted in dense, complex code [14].

Middlewares are an important feature of Express. Http requests to the server can be passed through any middlewares as specified by the route. This allows for functions that need to be performed on multiple routes to be gracefully separated from the individual request handlers. A good example of a middleware is an authentication middle as seen in Figure 3.3.

Figure 3.3 shows the authentication middleware used in Wiki Courses. (Note: the password in the jwt.verify function has been changed). This middleware uses the jsonwebtoken (jwt) package to track a user's authentication status. Every HTTP request that is trying to access a protected resource goes through this middleware to ensure they are only accessible by authenticated users. Creating this sort of middleware functionality in plain Node.js would require a lot of tedious and error-prone work and code. Express removes the



```
import { NextFunction, Request, Response } from "express";
import jwt from "jsonwebtoken";

export default (req: Request, res: Response, next: NextFunction) => {
  try {
    const token = req.headers.authorization;
    jwt.verify(token!, "secret_password");
    next();
  } catch (error) {
    res.status(401).json({ message: "You are not authenticated" });
  }
};
```

Figure 3.3: Authentication Middleware

need for this code to be handwritten by every individual developer by providing the ability to attach middlewares.

3.4 MongoDB

MongoDB acts as half of the database for Wiki Courses. It stores all of the information associated with user accounts and a small amount of information about courses. MongoDB is a NoSQL (Not only SQL) database commonly used in modern stacks like the MEAN stack. Wiki Courses uses MongoDB for four reasons: initial setup is easy and free; the database can exist in the cloud; it scales well as the application grows; and it has excellent integration with Node.js. Wiki Courses specifically uses MongoDB Atlas, which is the cloud-based version of MongoDB. This enables the server to access the database remotely and removes the need for any database software to exist on the machine hosting the website. The JavaScript package Mongoose is used to provide easy interaction between Node.js and MongoDB (see Figure 3.1).

SQL relational databases have been the standard for decades, though in recent years companies such as Google, Amazon, and Facebook have transitioned to NoSQL databases [14]. Performance, flexibility, and scalability are all frequently cited as reasons for this change [16]. SQL databases use a relational model that requires strict definitions for every table. NoSQL databases provide more flexibility by not adhering to the strict table structure of SQL while still including some relational capabilities, hence the acronym "Not only SQL" [14].

Mongoose is a JavaScript package used to interact with a MongoDB database from within Node.js. The structure of a MongoDB database resembles that of JSON (JavaScript Object Notation) documents, which makes it easy for JavaScript to work with the database. JSON is the conventional syntax for working with data in JavaScript [14]. Mongoose provides the connection to the remote database and provides functions for querying the database. Additionally, it allows for MongoDB documents to be defined in JavaScript using a Mongoose schema object. This allows for the documents in the database to be wholly defined in the server code.

3.5 Git

Git is a version control systems that is commonly used to store code for programming projects. Wiki Courses stores all of its server-side and client-side code in Git repositories. Specifically, Git repositories store versions of the contained files. When an update is made to the project it can be committed using the Git `commit` command. This creates a new version of the changed files. Previous versions are able to be "checked out" using the Git `checkout` command. This reverts the files in the repository to their state at a specified commit.

A novel aspect of Wiki Courses is its use of Git as a database. As a wiki, Wiki Courses can be edited by all users. This can cause problems when a user intentionally or accidentally alters the course in a negative way, such as by deleting a large portion of it. In order to recover from situations like this, the course data needs to be stored in a version control system so that it can be restored to earlier versions. There are many ways to create version control systems using a standard database like MongoDB. However, by putting course information in a Git repository and saving changes with Git commits, the courses automatically have versions saved every time the course is updated.

3.6 Amazon Web Services

Finally, for a website to be accessible on the internet, it needs to be hosted via a web hosting service. Amazon Web Services (AWS) provides a variety of simple hosting solutions, including free options for small websites. Wiki Courses is hosted using an Elastic Beanstalk AWS server. Elastic Beanstalk is a type of web server provided by Amazon that is simple to set up and scales well as applications grow. The API and SPA are hosted on separate Elastic Beanstalk servers.

Chapter 4 – Design & Implementation

The Wiki Courses website is separated into two sections: client-side and server-side. Each section is structured differently and uses different tools and technologies. The server-side manages a REST API along with both the MongoDB and Git databases, all of which is unseen by the end user. The client-side application manages the front-end single-page application, which includes a user interface that introduced visual design requirements not present in the server-side. This chapter will describe the implementation, structure, and design of each section of Wiki Courses.

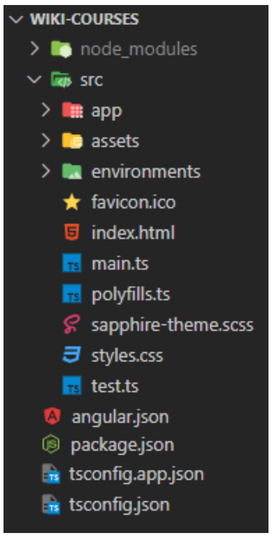
4.1 Client-Side

The code for the Angular application follows the file structure in Figure 4.1. The root folder mainly contains configuration files. Angular is written in TypeScript, a superset of JavaScript that must be transpiled into JavaScript to be run in the browser. The tsconfig files provide configuration information to the TypeScript transpiler. The package.json is key for any Node.js application. It defines all of the dependencies for the project as well as configuration details and scripts for testing and running the project.

The src folder holds the code for the application. The index.html file is the entry point for the SPA and main.ts is the entry point for the Angular code. These files are written by

Angular when a project is created and never changed. The source code that defines the application lies in the app folder.

When the application is transpiled using Angular's `ng build` command, a `dist` folder is output that contains the HTML, JavaScript, and CSS required to display the website. Angular has combined all of the code from the `src` folder in Figure 4.1 along with all of the dependencies into just a few files. These are the files that are actually hosted on the website. They are never modified directly and only change when the Angular application is rebuilt. This dist folder is created in a separate repository that contains a simple Node.js server. This server hosts the Angular SPA.



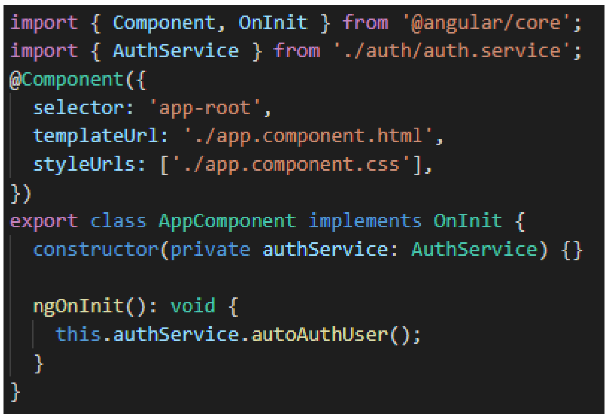
```
WIKI-COURSES
  > node_modules
  v src
    > app
    > assets
    > environments
      favicon.ico
      index.html
      main.ts
      polyfills.ts
      sapphire-theme.scss
      styles.css
      test.ts
    angular.json
    package.json
    tsconfig.app.json
    tsconfig.json
```

Figure 4.1: File Structure of Wiki Courses SPA Source Code

4.1.1 Angular Components

Components in Angular have three parts: an HTML template, a CSS bundle, and an Angular component object, which is a JavaScript class decorated with the component decorator as seen in Figure 4.2. Decorators are a special Angular syntax that allow a JavaScript class to be marked as an Angular object and provide metadata for using the object [22]. In Figure 4.2 the component decorator is used to specify three parts of the component: the selector (HTML tag), the HTML template to render when using the component, and a list of CSS files used to style the template. Component templates can range from very complex to very simple HTML. Components can include other components in their template, called component composition, which helps reduce the complexity of any single component template.

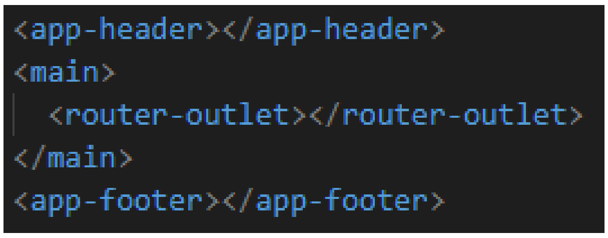
Angular begins at a root component that is rendered in the auto-generated index.html file. This component usually has very little logic in it and serves as a foundation for the application. In Figure 4.2 we can see the TypeScript code for the root component. The only thing it is doing is attempting to auto-authenticate the user if they have an active token. The HTML for the root component is similarly sparse as seen in Figure 4.3. It includes three special Angular components: app-header, app-footer, and router-outlet. The header and footer are custom components that make up the header of the website and the footer of the website. These are included in the root component's template since they should be displayed everywhere on the website. Finally, there is the router outlet that displays the appropriate component as defined by the Angular router.



```
import { Component, OnInit } from '@angular/core';
import { AuthService } from './auth/auth.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  constructor(private authService: AuthService) {}

  ngOnInit(): void {
    this.authService.autoAuthUser();
  }
}
```

Figure 4.2: Angular App-Root Component



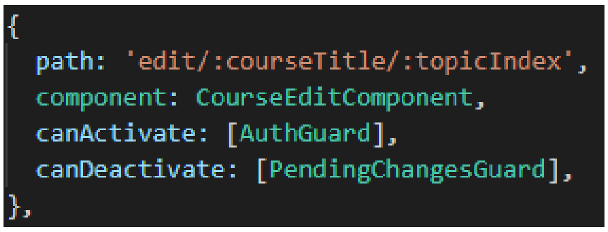
```
<app-header></app-header>
<main>
  <router-outlet></router-outlet>
</main>
<app-footer></app-footer>
```

Figure 4.3: Angular App-Root HTML Template

4.1.2 Angular Routing

Navigation in Wiki Courses is done via the Angular router. This router uses a *routes* object that defines pairs of paths and corresponding components. When a user navigates to a path in the routes object, the router loads the corresponding component in the router-outlet tag as seen in Figure 4.3. This prevents the browser from performing the navigation itself, which would reload the page. Instead, it only updates the content within this tag to reflect the appropriate component.

Wiki Courses leverages the *guards* feature of Angular routes. Guards are called by the router at a specified time during navigation. Figure 4.4 shows a route for the course edit page that has two guards: AuthGuard and PendingChangesGuard. AuthGuard runs whenever the router navigates to this page. It checks that the user is authenticated before allowing the router to navigate to the edit page. If the user is not authenticated then it redirects to the home page. PendingChangesGuard runs when the user attempts to leave the page. It checks if there are any unsaved changes and warns the user to save them before leaving the page or else they are lost. The ability to separate these guards from the pages on which they activate allows them to be reused for multiple routes and keeps all navigation logic within the router.



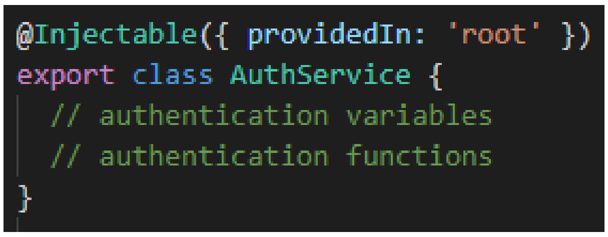
```
{
  path: 'edit/:courseTitle/:topicIndex',
  component: CourseEditComponent,
  canActivate: [AuthGuard],
  canDeactivate: [PendingChangesGuard],
},
```

Figure 4.4: Angular routing path with guards

4.1.3 Angular Services

In Wiki Courses, Angular services contain all the logic for making requests to the API. There is a separate service for every API entry point. These services are used by components for all interactions with the database. By separating this logic into services, all components can access the same functions. This avoids code repetition and isolates the responsibility of making HTTP requests.

Services in Wiki Courses are provided to components using the dependency injection function of Angular. Services are given the Injectable decorator (see Figure 4.5), that directs Angular to treat the service as a dependency. Angular applies the singleton design pattern to dependencies [23], which ensures that there is only one instance of the class [9]. All components that use the service will access the same instance.



```
@Injectable({ providedIn: 'root' })
export class AuthService {
  // authentication variables
  // authentication functions
}
```

Figure 4.5: Injectable decorator used in services

The use of a service can be seen in Figure 4.2. The service, in this case AuthService, is initialized in the component constructor where Angular dynamically injects the service and stores it in the specified variable when the component is first created. The syntax used in the constructor is shorthand for creating an instance variable named authService of type AuthService, having an AuthService instance passed into the constructor, and assigning the provided AuthService to the instance variable. The root component then uses the service when the component is initialized to attempt to auto-authenticate users if they are already logged in.

4.1.4 Design & Interface

Wiki Courses is intended to host a variety of different courses, all of which use the same style interface. The content of the courses is customizable by the users but the interface through which that content is accessed is not. This is to provide continuity between courses so that users know what to expect when they visit a course. This is a principal followed by most wikis: each subject should follow the same format.

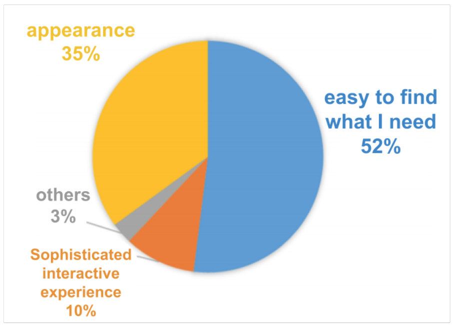


Figure 4.6: The Most Important Factors in Web Design. Reprinted from [21], page 19.

Research on user preferences in interface design was consulted to determine an appropriate style for Wiki Courses. A study titled *Does Color Matter on Web User Interface Design?* gathered statistics on students' views of "the most important factor in designing websites" [21]. Figure 4.6 shows that the ability to find what they are looking for is by far the most important aspect of a website. Appearance was second-most important, and sophisticated interactive experience was only chosen by 10% of the surveyed students [21]. With this in mind, Wiki Courses emphasizes simplicity in design to appeal to student preferences with the goal of supporting user retention. Each course maintains the same navigation and layout so that it is easy to switch between courses without confusion.

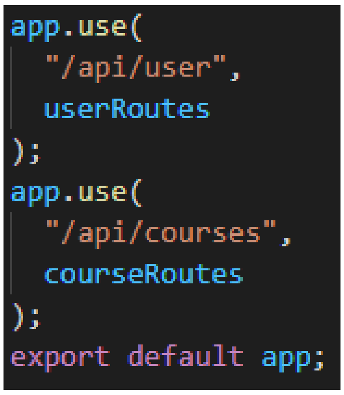
Color scheme was also considered in the design of Wiki Courses. Table 4.1 shows the emotional effects of colors as viewed from web browsers. Other studies have shown specific moods colors can evoke. For instance, red can evoke importance and green can engender stability within a viewer [21]. This research on the effect of colors informed the decision-making process when designing Wiki Courses. Light blue and white were selected to be the main color scheme for the website since they promote feelings of safety, calm, openness, simplicity, cleanliness, and virtue [21].

Color	Promotes
Red	Importance, power, youth
Orange	Uniqueness, friendliness, arise energy and a sensation of movement
Yellow	Happiness, enthusiasm, antiquity (darker shades)
Green	Growth, stability, financial themes, and environmental themes
Blue	Safety, calm, openness (lighter shades), strength and reliability (darker shades)
Purple	Luxury, romance (lighter shades), mystery (darker shades)
Black	Power, edginess, sophisticated and timeless
White	Simplicity, cleanliness, virtue
Gray	Formality, neutrality, melancholy
Ivory	Elegance, simplicity, comfort
Beige	Traits of surrounding colors, humility, a secondary or background color

Table 4.1: Color Emotions on Web UI Design. Reprinted from [21], page 18

4.2 Server-Side

The server begins with app.js, the file that creates the server using Express, and exports it in order to be served by the server.js file. The Express object *app* is created with the express() constructor that simplifies the process of creating an application in Node.js. This file also connects to the remote MongoDB database using Mongoose, sets Cross-Origin-Resource-Sharing (CORS) headers, and finally serves the API endpoints with the code in Figure 4.7. The app.use() method allows the application to connect a URL path to an express.Router object.



```
app.use(
  "/api/user",
  userRoutes
);
app.use(
  "/api/courses",
  courseRoutes
);
export default app;
```

Figure 4.7: Building the Express Object

When the API is accessed at a valid path, `/api/user` or `/api/courses` as shown in Figure 4.7, Express passes the request through to the router responsible for processing requests at that endpoint. The router looks at the path of the request, the type of request (put, get, delete, etc), and any parameters to the request. It then selects a set of middlewares to pass the request through and a function to respond to the request. Figure 4.8 describes this entire process. In order to separate the responsibility of responding to requests from routers, controller files are created. Typically each router has a corresponding controller that contains all the functions that the router might use to respond to a request.

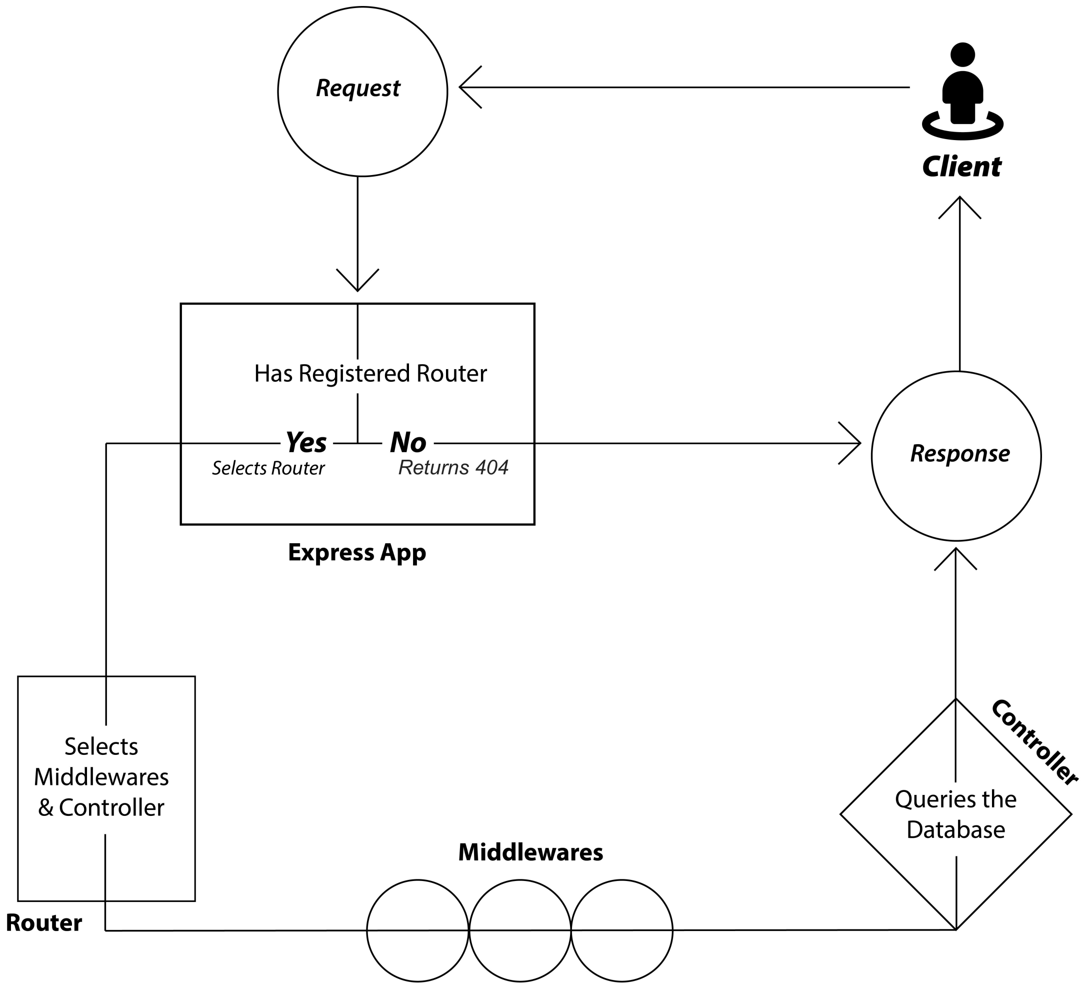
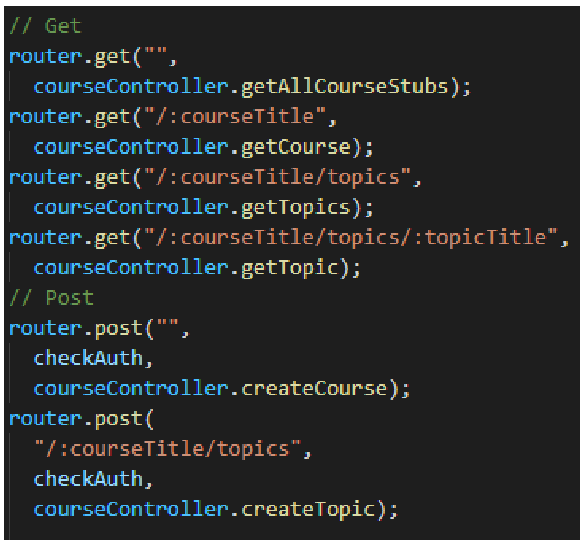


Figure 4.8: Express Decision-Making Process

4.2.1 API

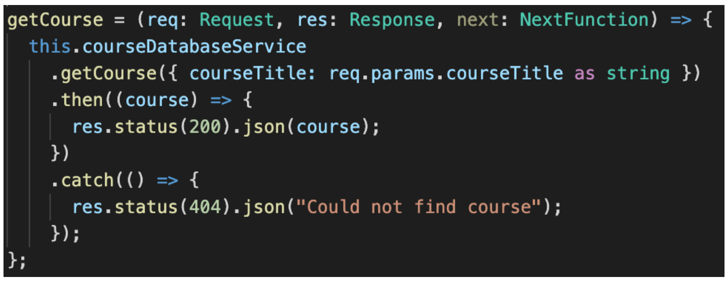
The Wiki Courses API has two primary entry points as seen in Figure 4.7: *user* and *courses*. Both entry points have a set of defined routes that describe what action should be taken given a particular HTTP request. The routes object connects an HTTP request with a function from the corresponding controller. Portions of a path that begin with a colon are parameters and can contain any value. For example, in Figure 4.9, a get request to the /course route with one parameter is passed to the getCourse function of the CourseController. The parameter will be stored in a variable named courseTitle.



```
// Get
router.get("",
  courseController.getAllCourseStubs);
router.get("/:courseTitle",
  courseController.getCourse);
router.get("/:courseTitle/topics",
  courseController.getTopics);
router.get("/:courseTitle/topics/:topicTitle",
  courseController.getTopic);
// Post
router.post("",
  checkAuth,
  courseController.createCourse);
router.post(
  "/:courseTitle/topics",
  checkAuth,
  courseController.createTopic);
```

Figure 4.9: get and post handlers for courses

The controller contains the logic to handle the request and return the appropriate response. This usually involves performing a query on the database. In the getCourse function in Figure 4.10, the courseDatabaseService is used to query a course with a particular title, which it accesses using the params property of the request. After performing the query, it creates a response that is sent to the user containing either the query result or an error message if the query failed.

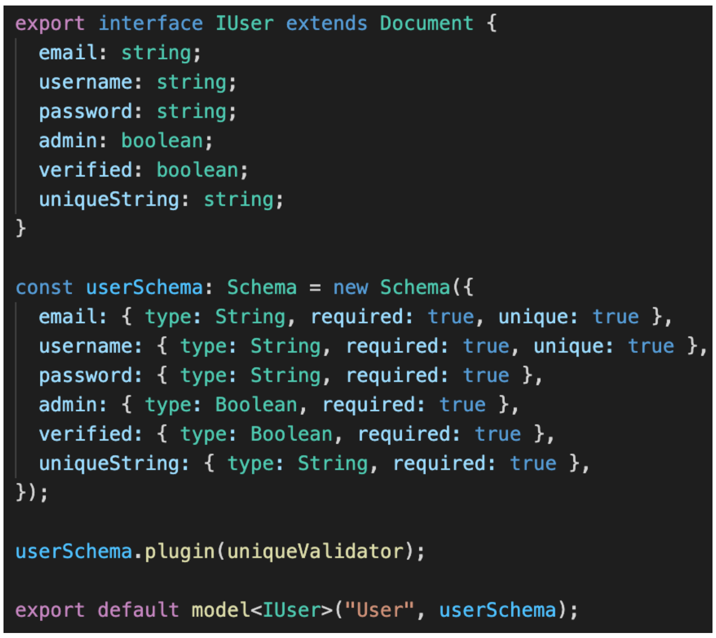


```
getCourse = (req: Request, res: Response, next: NextFunction) => {
  this.courseDatabaseService
    .getCourse({ courseTitle: req.params.courseTitle as string })
    .then((course) => {
      res.status(200).json(course);
    })
    .catch(() => {
      res.status(404).json("Could not find course");
    });
};
```

Figure 4.10: getCourse function in CourseController

4.2.2 MongoDB

The MongoDB database is hosted on the cloud using MongoDB Atlas. It is not actually found in the code for the website, nor is it on the AWS machine hosting the website. The server uses the JavaScript package *Mongoose* to connect to the database and interact with it. The server creates models using a Mongoose schema (Figure 4.11) that define the structure of documents in the database.



```
export interface IUser extends Document {
  email: string;
  username: string;
  password: string;
  admin: boolean;
  verified: boolean;
  uniqueString: string;
}

const userSchema: Schema = new Schema({
  email: { type: String, required: true, unique: true },
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  admin: { type: Boolean, required: true },
  verified: { type: Boolean, required: true },
  uniqueString: { type: String, required: true },
});

userSchema.plugin(uniqueValidator);

export default model<IUser>("User", userSchema);
```

Figure 4.11: Mongoose User Schema and related IUser interface

Since the server code is written in TypeScript, a superset of JavaScript, there is an extra step of creating an interface to model the structure of the data when it is returned from the database. A Mongoose schema is used to interact with the database and define models within the database, but when a query is performed, the result is a simple JavaScript object, not a schema object. By creating the IUser interface to go along with the User schema, the model can be defined as returning an IUser object. This allows any code that interacts with the schema to know the exact structure of the object returned from it. This is useful when programming as it enables IDEs (Integrated Development Environments) to check whether the returned objects are being used in a valid manner. For example, if a user document was queried and then the querying function accesses the *name* property of the user; the IDE is able to highlight this as an error since there is no name property on the return type. Without the interface there would be no way for the IDE to know the structure of the returned user as it would be an `any` type by default.

One of the key features of Mongoose is object mapping. Object mapping refers to how Mongoose translates data from the format in MongoDB to JavaScript objects. This process can be seen in Figure 4.12. Object mapping allows data in MongoDB to be created and manipulated within JavaScript without the need to understand how MongoDB actually stores the data. Additionally, with the use of TypeScript interfaces, the Node.js server is able to know the exact structure of the returned data, still without knowing anything about the representation of the data in MongoDB.

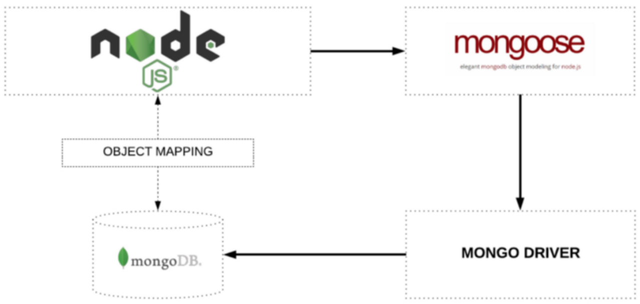


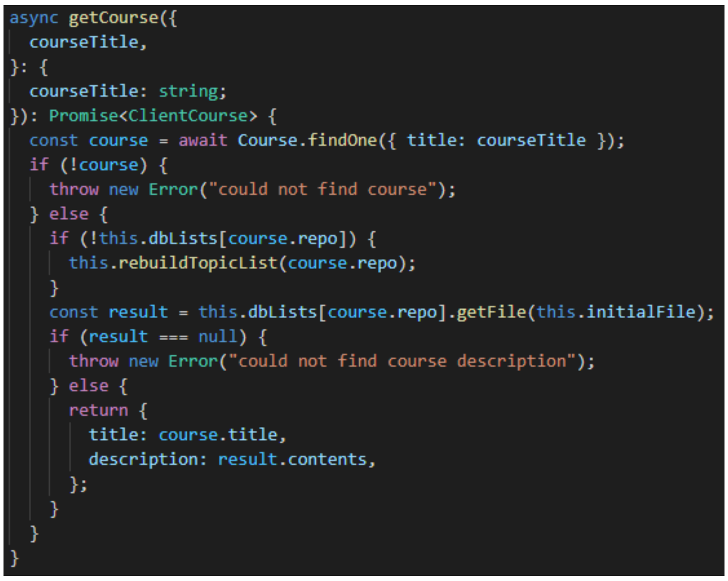
Figure 4.12: Object Mapping Between Node and MongoDB Managed by Mongoose. Reprinted from [13].

4.2.3 Git Database

Wiki Courses uses Git as a database alongside MongoDB. Every course on the website has its own Git repository on the server. MongoDB stores the title of each course along with the path to the corresponding repository. The content of the topics of the course are stored in HTML files in this repository. The title of each topic is the name of the corresponding HTML file. When a query is made for a particular course, the server first checks MongoDB for the path to the course repository while also verifying that the course exists. If the course exists then the server reads in the files from the repository for that course and constructs JavaScript objects with the title and content of each topic that is then sent to the client.

The server has a special service for handling this hybrid database. The service is called *courseDatabaseService* and its use is seen in Figure 4.10. By separating these functions into their own service, the controllers do not need to handle the logic for using the MongoDB database and the Git database. The function called in Figure 4.10 can be seen in Figure 4.13.

One problem encountered by this approach is the ordering of topics within a course. The topics of a course have a specific order, but since the topics exist as files in a folder, the order of topics is not saved. Additionally, these topics need to be able to be re-ordered and maintain their order when a topic is deleted. For the database to maintain the correct order of topics, it uses a linked list data structure. Each node in the linked list contains the path to a particular topic file along with a pointer to the topic that comes after it and a pointer to the topic that comes before it. The courseDatabaseService maintains a set of these structures in memory, one for each course. The use of this structure can be seen in Figure 4.13 where the linked list of topics for the given repository is accessed at `this.dbLists[course.repo]`.

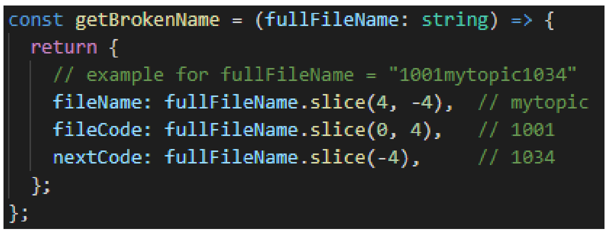


```
async getCourse({
  courseTitle,
}: {
  courseTitle: string;
}): Promise<ClientCourse> {
  const course = await Course.findOne({ title: courseTitle });
  if (!course) {
    throw new Error("could not find course");
  } else {
    if (!this.dbLists[course.repo]) {
      this.rebuildTopicList(course.repo);
    }
    const result = this.dbLists[course.repo].getFile(this.initialFile);
    if (result === null) {
      throw new Error("could not find course description");
    } else {
      return {
        title: course.title,
        description: result.contents,
      };
    }
  }
}
```

Figure 4.13: getCourse function of the courseDatabaseService

Since this data structure resides in memory it must be able to be constructed from a folder whenever the server restarts. In Figure 4.13 the function checks if the linked list has been created for this repo already. If it has not, it calls the rebuildTopicList function before proceeding. This function rebuilds the topic list from the repository folder. In order to do this, the linked list gives every file a four-digit code that is appended to the file name. It also appends the code of the next topic at the end of the file name. Together, all three parts of the filename are called a *brokenName* (Figure 4.14). When the rebuildTopicList function is run, it constructs the linked list from a folder by reading in the file names, splitting them into broken names, and using the codes to order the list. The nodes are then responsible for returning the correct file name without the codes when the topic data is queried.

The purpose of using Git is to leverage its version control system. Wikis require some sort of version control system due to the lack of oversight of edits. Most wikis have some sort of written version control system that utilizes a standard database. These systems work well but they are very time consuming to create from scratch. The benefit of using Git is that a version control database can be implemented very quickly as Git has built in version control. Within the scope of this thesis, Git was an optimal choice for this reason.



```
const getBrokenName = (fullFileName: string) => {
  return {
    // example for fullFileName = "1001mytopic1034"
    fileName: fullFileName.slice(4, -4),  // mytopic
    fileCode: fullFileName.slice(0, 4),   // 1001
    nextCode: fullFileName.slice(-4),     // 1034
  };
};
```

Figure 4.14: brokenName structure in the linked list nodes

When a user requests to revert a course to a previous version, the server executes a Git `checkout` command for the commit corresponding to the requested version. The versions displayed to the user are the output of a Git `log` command and show the username of the editor, the timestamp, and the message provided by the user who made the change. After a reversion, the later commits have not been erased, so they can still be checked out, but they are no longer displayed to users on the website.

When a course is deleted, the document in MongoDB containing the name of the course and the repo location is removed. However, the course folder in Git is not deleted. This allows an admin user to restore the course if necessary by manually adding the course back into MongoDB with the path to the course folder.

Figure 4.15 shows how requests are processed in the database system and the responsibilities of each part. The requests are received by the course controller which determines what sort of database action is required and then calls the course database service to perform that action. The course database service fulfills these requests by using the git database service to perform git commands, the linked file list service to extract topic information from the repositories, and MongoDB to get repository names.

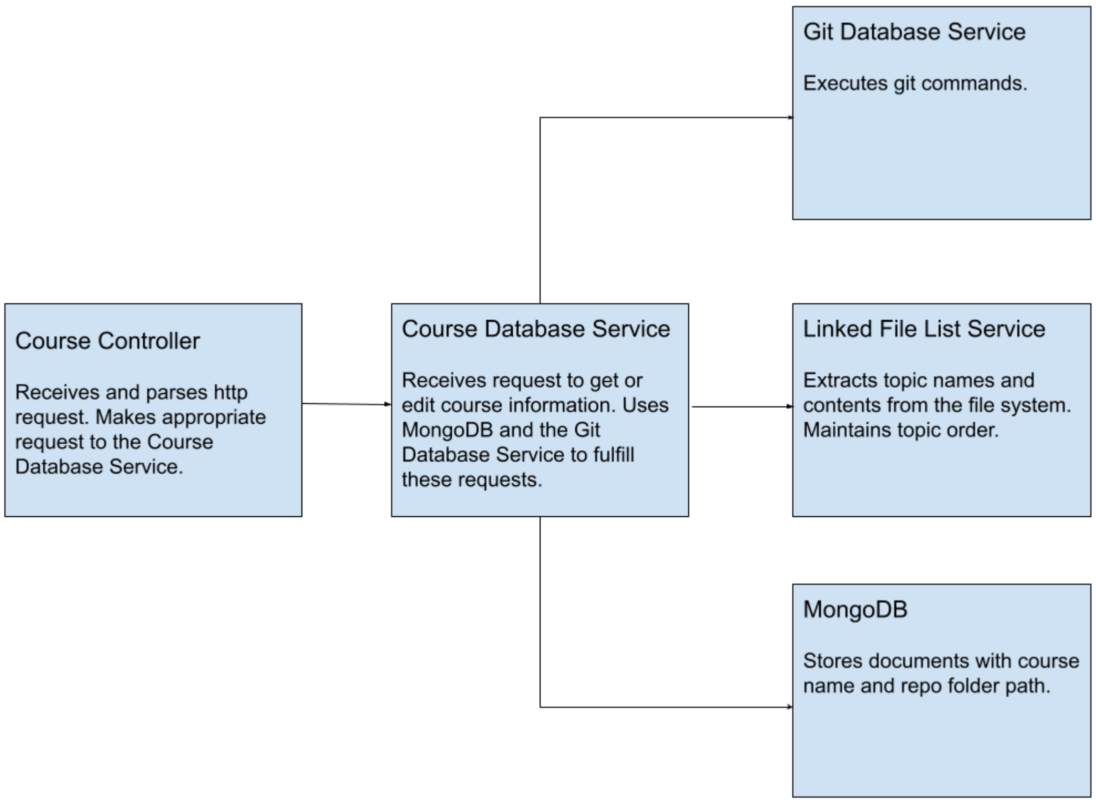

Git Database Service
Executes git commands.
Course Controller
Receives and parses http request. Makes appropriate request to the Course Database Service.
Course Database Service
Receives request to get or edit course information. Uses MongoDB and the Git Database Service to fulfill these requests.
Linked File List Service
Extracts topic names and contents from the file system. Maintains topic order.
MongoDB
Stores documents with course name and repo folder path.

Figure 4.15: Course Database System

Chapter 5 – Testing

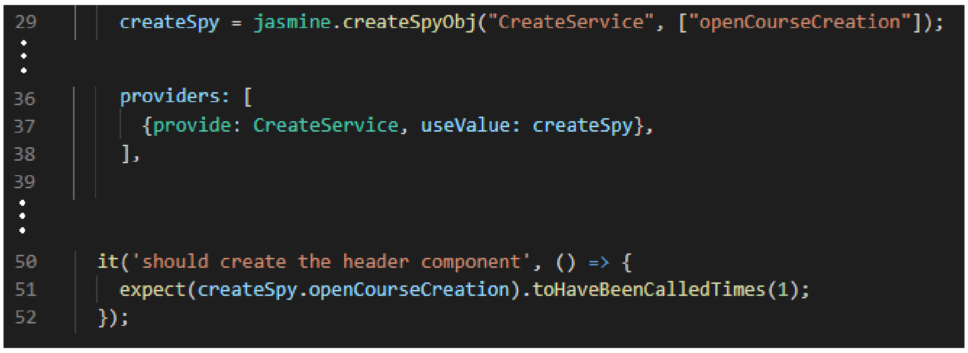
Automated tests are an important part of developing any large scale application. The web pages, API endpoints, and internal functions of the website all need to be tested to ensure they are running correctly. Wiki Courses was tested using JavaScript test functions. Since the API and SPA were built with different JavaScript frameworks, they are also tested with different tools. Angular has a built-in unit testing framework using the tools Karma and Jasmine. This testing framework is ideal for Angular applications. The Angular testing framework does not provide end-to-end testing functions. Since the Wiki Courses API is not an Angular application, it required a different tool for testing. The API was tested using the JavaScript testing runner and framework Jest, which is particularly suited for testing Node.js APIs. Jest is used for both unit tests and end-to-end tests on the API.

5.1 Angular SPA Testing

The Wiki Courses Angular SPA was tested using Karma and Jasmine. Karma is a test runner for JavaScript applications. It creates a Node.js server and runs source code against test code on a number of specified browsers [25]. This allows code to be tested on real browsers in the same manner that it will be used by clients. Wiki Courses was tested in Google Chrome, the default testing browser for Angular applications in Karma. Karma does not provide an assertion library or any kind of testing framework. Testing frameworks provide JavaScript functions to write tests for JavaScript applications. However, they do not provide a utility to run these tests. This utility is provided by test runners such as Karma.

Jasmine is a behavior-driven development (BDD) JavaScript testing framework [4]. BDD is a development paradigm that encourages quick collaboration between the developers and non-developers working on a project. BDD frameworks define tests in a way that is easily understandable to those without technical knowledge. Wiki Courses was developed by a single individual and therefore did not employ the BDD methodology. However, Jasmine was designed to support BDD but not require it. It is a strong and useful testing framework irrespective of development methodology [4].

Jasmine is used for unit testing where small sections of source code are tested individually. It is not used to perform integration testing or end-to-end testing. Jasmine provides a structure for tests along with numerous testing functions and utilities. In order to test components in isolation, dependencies for those components must be faked. Jasmine provides Jasmine spies to fake dependencies. In addition to providing mock dependency functions, spies record information about function calls to test if the dependencies are being used correctly [4].



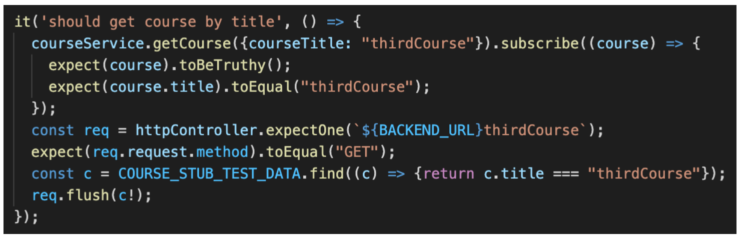
```
29      createSpy = jasmine.createSpyObj("CreateService", ["openCourseCreation"]);
⋮
36      providers: [
37        {provide: CreateService, useValue: createSpy},
38      ],
39
⋮
50    it('should create the header component', () => {
51      expect(createSpy.openCourseCreation).toHaveBeenCalledTimes(1);
52    });
```

Figure 5.1: Example of a Jasmine spy being used to mock the CreateService

Figure 5.1 shows the creation of a Jasmine spy. The spy is created on line 29 using the jasmine.createSpyObj function, which is passed the name of the service it is providing and a list of functions for that service. The resulting spy provides the CreateService and has one function: openCourseCreation. Then, on line 37, the spy is provided to the Angular component in place of the normal CreateService. Finally, the spy is used in a test where it checks that the openCourseCreation method has been called one time on line 51. Jasmine spies were heavily used in the testing of Wiki Courses.

Angular has testing functionality built into it using Karma and Jasmine [24]. Wiki Courses used the default configurations for both of these tools. Angular provides the ’ng test’ console command to run tests. In addition to launching Karma, ng test also builds the necessary parts of the Angular application for the tests. Angular also provides some additional testing functions that are specific to Angular applications and work in tandem with Jasmine functions.

Unit tests require code to be tested in isolation. To do this, any code that is depended upon by the code being tested must be replaced with a fake version. Angular provides fake versions of common Angular services such as the HttpClient. These fake versions work similarly to Jasmine Spies with some additional features. In Figure 5.2, the GetCourse method of the CourseService is being tested. In order to prevent actual HTTP requests being made to the API, an Angular HttpTestingController (httpController) is used in place of the normal HttpClient. The httpController provides data about requests made to it which is used to test that the CourseService makes the correct HTTP requests.



```
it('should get course by title', () => {
  courseService.getCourse({courseTitle: "thirdCourse"}).subscribe((course) => {
    expect(course).toBeTruthy();
    expect(course.title).toEqual("thirdCourse");
  });
  const req = httpController.expectOne(`${BACKEND_URL}thirdCourse`);
  expect(req.request.method).toEqual("GET");
  const c = COURSE_STUB_TEST_DATA.find((c) => {return c.title === "thirdCourse"});
  req.flush(c!);
});
```

Figure 5.2: Jasmine test for the Course Service

The test in Figure 5.2 begins by calling the getCourse method on the courseService.

Since this method is asynchronous, it sets up a callback function that will run when the method completes. This callback function asserts that the correct course was returned. When the getCourse method runs, it makes an HTTP request to the API for a particular course. During testing, this request is made to an HttpTestingController rather than the normal HTTP service. After making the call to getCourse, the test expects that the httpController has received an HTTP request for a particular endpoint. After checking that the correct request was made, the httpController returns a hardcoded course value to the getCourse method using the flush function. Once this value is returned, the getCourse method will complete and the assertions in the callback function will run.

Each unit test in Wiki Courses tests a specific part of a component. The quantity of tests performed on each component of Wiki Courses depends upon the complexity of that component. For example, the footer of the website displays a small amount of information and contains a link to the feedback page. In contrast, the header contains many links which display conditionally depending on the login status of the user. Additionally, some of these links run functions rather than navigate the user to a new page. As a result, the header component requires many more tests than the footer component.

These tests can be roughly divided into two categories: structural and functional. Structural tests check the HTML of the component and compare part of it to an expected value. Functional tests check that the logic of the component behaves correctly. An example of a structural test would be a test that looks at a course page and asserts that the title of the course should be displayed on the page. An example of a functional test could be a test that asserts that a PUT HTTP request should be made when the ’save course’ button is clicked while editing a course. Some tests contain both structural and functional elements. These tests check whether the HTML of a component changes correctly when a certain function is performed.

Table 5.1 shows the number of unit tests created for each of the primary components of Wiki Courses. The components are split into 3 categories: page, element, and service. Pages are Angular components that make up an entire web page and are routed to via the Angular router. Elements are Angular components that are used in pages but are not entire pages themselves. Services are libraries of functions and do not have any HTML associated with them. Therefore tests on services are all functional. The table orders the components by number of associated tests. This ordering also shows the comparative complexity of each component.

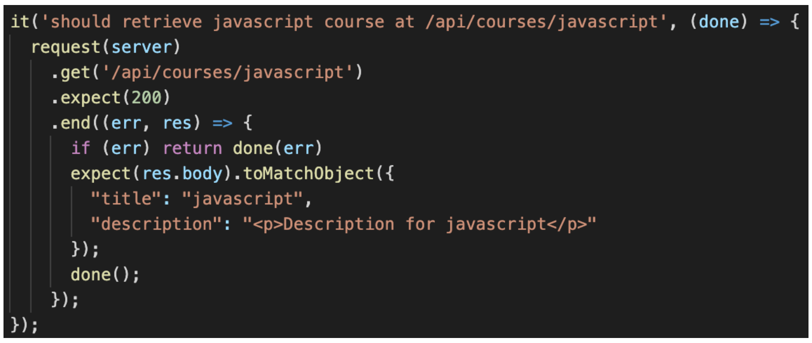
Component Name	Component Type	Number of Tests
Course Service	Service	17
Course Edit	Page	13
Course Display	Page	11
Header	Element	11
Auth Service	Service	9
Advanced	Page	8
Course List	Page	6
Topic Menu	Element	5
Account	Page	5
Signup	Page	4
Login	Page	4
Verified	Page	4
Footer	Element	3
About	Page	2

Table 5.1: Number of SPA unit tests per component, all performed in Jasmine and Karma

5.2 API Testing

The Wiki Courses API was tested using Jest. Jest is a widely used JavaScript test runner with many built-in testing features. It supports testing of asynchronous operations, which is important when testing APIs. Jest includes an assertion function library that is built on Jasmine [11]. Jest is very similar to Karma and Jasmine used together as it provides both a testing framework and a test runner. However, it includes more functions for testing an API. Jest also works well with Node.js servers that use TypeScript while many other testing frameworks only support JavaScript. This makes Jest an excellent choice for testing Wiki Courses as Wiki Courses is a Node.js API written in TypeScript.

Figure 5.3 shows a test for the API endpoint used to retrieve a single course by title. Jest starts the server and visits the specified URL. It then compares the response code and response body to values provided in the test. Wiki Courses uses this method of testing for all of its API endpoints.



```
it('should retrieve javascript course at /api/courses/javascript', (done) => {
  request(server)
    .get('/api/courses/javascript')
    .expect(200)
    .end((err, res) => {
      if (err) return done(err)
      expect(res.body).toMatchObject({
        "title": "javascript",
        "description": "<p>Description for javascript</p>"
      });
      done();
    });
});
```

Figure 5.3: Jest test for /api/courses/javascript endpoint

API Endpoint	Accepted request types	# Tests
/api/courses/:courseTitle	GET, DELETE	5
/api/courses/:courseTitle/topics/:topicTitle	GET, DELETE	5
/api/courses/:courseTitle/topics	GET, POST	4
/api/courses/:courseTitle/title	PUT	4
/api/courses/:courseTitle/topics/:topicTitle/title	PUT	4
/api/courses/:courseTitle/description	PUT	3
/api/courses/:courseTitle/topics/:topicTitle/contents	PUT	3
/api/courses	GET, POST	3

Table 5.2: API tests per endpoint, all performed in Jest

Table 5.2 shows the number of tests for each course API endpoint. Every course API endpoint is tested at least once. Many are tested in multiple ways by accessing them with different request bodies and headers. The tests check for both success and failure by issuing valid requests and invalid requests. Endpoints that accept multiple types of requests (GET, POST, PUT, DELETE) have more tests than endpoints that only accept one type of request.

Manual testing of the entire website, SPA and API together, was also performed. These tests are not as rigorous as automated tests and are only intended to supplement them. Automated tests will only catch errors that the developer checks for. There can be unexpected errors that are missed by automated tests but can be caught in a manual test. When performing a manual test, all pages of the website are visited both when logged out and logged in. The content of these pages is verified by the tester. Manual tests were performed regularly during the development of Wiki Courses and often found errors missed by automated tests.

Chapter 6 – Conclusion

Wiki Courses successfully demonstrated the use of Git as a back-end database in implementing a special purpose wiki. The use of Git provided a version-control system for free, which made the implementation much faster than it would be if implementing a version-control system from scratch on top of a standard database. The overall implementation process went smoothly, with no area indicating particular difficulty. While specific performance tests were not run, the apparent performance of Git as a backend database subjectively seemed to be sufficient. Wiki Courses performed well under the small scale load tested, and the end user could observe no indication about what was being used on the backend.

Implementing the Wiki Courses database this way required no knowledge of database management systems or of SQL. It did require an understanding of Git commands, something that is nearly ubiquitous among today's developers. Little difficulty was experienced in choosing the methodology for mapping the chosen schema onto the names stored in Git. If the schema had been more complex or subject to frequent change then this area may have presented more drawbacks to this approach.

The primary downside to using Git as a backend is the loss of generality in approaching the data as if it were stored in a database. It could become complex to decide how to map individual objects and their relationships to one another into file names in Git. It would also be more difficult to update the schema, although this is not always simple with standard databases either. The use case of a very large number of inter-related records with many different types of objects and a high volume of of accesses and searches is not the intended use case that was addressed here.

Wiki Courses itself, as a content creation platform, worked as hypothesized. Multiple creators were able to create courses and update them with no difficulty. While formalized scoring of user research was not conducted, users did report finding the interface intuitive and were able to create content quickly and without issue. They did feel that the editing capabilities could be enhanced to allow for more flexibility in content creation.

6.1 Future Work

Future work for Wiki Courses includes larger scale use of Git as a database to see how it performs with significantly more content. Performance tests will require a version control database that uses standard database technology such as MongoDB for comparison. These tests were not feasible in the scope of this thesis due to the time consuming nature of creating a version control database. However, these tests are critical in determining how effective a Git content database is.

Wiki Courses has the potential to be an effective website if certain improvements are made and more user feedback is acquired. Some of these improvements are as follows:

- **More types of content**
  Text lectures could be supplemented with additional content such as images, videos, quizzes, and downloadable content. These additional types of media would greatly improve the quality of lectures by providing editors with more options to best convey information.

- **Improved version management**

  The current system does not allow editors to view what a previous course version looked like before reverting to it. This functionality would be a big improvement as it would reduce the importance of editors providing detailed messages about their changes. Additionally, non-admin editors could also be given the ability to revert a course to a newer version rather than only being able to revert to an older version.

- **Increased moderation**

  Malicious edits can be reduced by requiring changes to be approved by multiple editors. Users can also be ranked by how much they contribute and some privileges could be restricted to higher ranked users. Machine learning software can also be employed to detect edits that appear malicious and flag them for review.

- **Improved account pages**

  Account pages could allow users to save courses they are working on and track the edits they have made to courses. They could also allow users to change their username and email as well as recover their password. Other user information, such as rank, could be provided on the account page as it is added to the website.

- **User Testing and Feedback**

  User feedback would help evaluate the quality and potential usefulness of the website. It would also provide information on what improvements and features are most important to implement.

Wiki Courses is a pilot proof of concept website for an academic course wiki. It also demonstrates how Git can be used as a version control database. Wikis have proven to be a powerful type of website that effectively crowd-source information. While this does not guarantee that courses created in a wiki would be of a high quality, it is an indicator of the possibility.

Bibliography

[1] Sanchit Aggarwal and Jyoti Verma, *Comparative Analysis of MEAN stack and MERN Stack*, International Journal of Recent Research Aspects **5** (2018), no. 1, 133–137.

[2] June Ahn, Cindy Weng, and Brian S. Butler, *The dynamics of open, peer-to-peer learning: What factors influence participation in the p2p university?*, 2013 46th hawaii international conference on system sciences, 2013, pp. 3098–3107.

[3] Valentin Bojinov, *RESTful Web API Design with Node.js*, 2nd ed., Packt Publishing, Birmingham, UK, 2016.

[4] Jess Campbell, *Jasmine JS: Start Testing From Scratch*, 'https://www.testim.io/blog/jasmine-js-a-from-scratch-tutorial-to-start-testing' (2021).

[5] CERN, *LHCb*, 'https://home.cern/science/experiments/lhcb' (2022).

[6] Marco Clemencic, *A git-based conditions database backend for lhcb*, 23rd international conference on computing in high energy and nuclear physics (chep 2018), 2019, pp. 3098–3107.

[7] Joseph Corneli and Alexander Mikroyannidis, *Crowdsourcing education on the web: A role-based analysis of online learning communities*, 2012.

[8] Parker Flynn-Adey, *Using Wikis to Support Student Inquiry in Large Math Classes*, The Electronic Journal of Mathematics and Technology **15** (2021), no. 2.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, 1st ed., Addison-Wesley, Boston, MA, 1995.

[10] Bill Hunter, *Reflections on the Co-Creation of a Course Wiki*, Canadian Journal of Action research **18** (2017), no. 3, 2–11.

[11] Zhongyuan Jin, *Design and implementation of full-stack testing for web spa in javascript*, Ph.D. Thesis, Aalto University, 2020.

[12] Gerald C. Kane and Robert G. Fichman, *The Shoemaker's Children: Using Wikis for Information Systems Teaching, Research, and Publication*, MIS Quarterly **33** (2009), no. 1, 1–17.

[13] Nick Karnik, *Introduction to Mongoose for MongoDB*, 'https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57' (2018).

[14] Jason Krol, *Web Development with MongoDB and Node.js*, 1st ed., Packt Publishing, Birmingham, UK, 2014.

[15] Teemu Leinonen, Tere Vadén, and Juha Suoranta, *Learning in and with an open wiki project: Wikiversity's potential in global capacity building*, First Monday **14** (2009), no. 2.

[16] Michael Madison, Mark Barnhill, Cassie Napier, and Joy Godin, *NoSQL Database Technologies*, Journal of International Technology & Information Management **24** (2015), no. 1, 1–13.

[17] Emily Jane McTavish, Cody E. Hinchliff, James F. Allman, Joseph W. Brown, Karen A. Cranston, Mark T. Holder, Jonathan A. Rees, and Stephen A. Smith, *Phylesystem: a git-based data store for community-curated phylogenetic estimates*, Bioinformatics **31** (2015), no. 17, 2794–2800.

[18] Microsoft, *What is Git?*, 'https://docs.microsoft.com/en-us/devops/develop/git/what-is-git' (2021).

[19] Victor Ofoegbu, *The Only NodeJs Introduction You'll Ever Need*, 'https://codeburst.io/the-only-nodejs-introduction-youll-ever-need-d969a47ef219' (2018).

[20] Vinci Rufus, *AngularJS Web Application Development Blueprints*, 1st ed., Packt Publishing, Birmingham, UK, 2014.

[21] Wirania Swasty and Andreas Rio Adriyanto, *Does Color Matter on Web User Interface Design?*, CommIT (Communication & Information Technology) Journal **11** (2017), no. 2, 17–24.

[22] Angular Team, *Component*, ‘https://angular.io/api/core/Component#description’ (2020).

[23] ______, *Injectable*, ‘https://angular.io/api/core/Injectable’ (2020).

[24] ______, *Testing*, ‘https://angular.io/guide/testing’ (2021).

[25] Karma Team, *Karma Official Documentation*, ‘http://karma-runner.github.io’ (2021).

[26] Wikiversity, *Wikiversity*, Wikiversity.org (2020).

Vita

Willow Emmeliine Sapphire was born in 1994 in Raleigh, North Carolina to Barton and Pamela Vashaw. Computers and code were a large part of their childhood. They graduated with an M.S. in Computer Science from Appalachian State University. They have worked as a full stack web developer, a mobile app developer, and a web developer throughout their education, both independently and for the University-affiliated BeeInformed Partnership. They aspire to continue within academia and attain a PhD to teach at the university level.