



What To Cook (WTC)

Project Engineering

Year 4

Denis Costello

Bachelor of Engineering (Honours) in Software and
Electronic Engineering

Atlantic Technological University

2023/2024



Figure 1-1 – What To Cook

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University of Galway.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____ Denis Costello _____

Acknowledgements

I would like to thank my supervisor, Brian O'Shea, for his assistance and guidance throughout the duration of my final year project.

I would also like to thank my Project Engineering lecturers, Michelle Lynch, Paul Lennon, Niall O'Keeffe, and Ben Kinsella, who have been there to provide guidance whenever it was required.

Table of Contents

1	Summary	7
2	Poster	8
3	Introduction	9
4	Tools and Technology.....	10
4.1	Microservice Architecture	10
4.2	Spring Boot.....	10
4.2.1	Spring Boot Authentication	10
4.2.2	Spring Cloud OpenFeign	10
4.2.3	Spring Boot Email	10
4.3	Next.js	10
4.4	MongoDB	11
4.5	MongoDB Atlas.....	11
4.6	Scrapy.....	11
4.7	Cloud Service (AWS / Kubernetes)	Error! Bookmark not defined.
5	Project Architecture	12
6	Project Plan	13
7	Project Code.....	14
7.1	Front-end	14
7.1.1	Pages folder.....	14
7.1.2	Components	16
7.1.3	Middleware	21
7.1.4	Context API.....	22
7.2	Backend.....	23

7.2.1	Authentication Service	23
7.2.2	Add Food Service	27
7.2.3	Get Food Service	27
7.2.4	Recipe Service	28
7.2.5	Email Service	30
7.3	Web Scraping	31
8	Ethics	34
9	Conclusion	35
10	References.....	36

1 Summary

The goal of this project is to reduce food waste. What To Cook ensures that the user is using all their food. It is estimated in Ireland, that we waste 800,000 tons of food each year [1]. What To Cook will help the user mealtime both quicker and more convenient. It is more cost-effective as less food is being wasted.

The project scope is focused on the web application, with the main deliverable being the recommendation of recipes based on the food inputted by the user through the application.

Some important features of this project are the Next.js front-end application that allows the user to input their food by typing, uploading a file, or using the device's camera to take a picture. This project uses Spring Boot microservices for scalability, breaking the application into smaller, independent services, and improving performance.

The main technologies used in this project were Java, Spring Boot, Spring Security, JWT, Docker, JavaScript, Next.js, Clarifai API, and Python web-scraper framework Scrapy. JavaScript and Next.js are used in the front-end. Next.js integrates the Clarifai API food-item-recognition AI model for food recognition, providing a convenient way for users to add their ingredients to the application. Spring Boot microservices are used in the backend for scalability, breaking the application into smaller, independent services, and improving performance. The microservices connect to Mongo Atlas for storing data and Mongo Atlas search index. Scrapy is used to gather recipes from the BBC Good Food website. The data is stored in a MongoDB Atlas database.

The project allows the user to sign up or sign in. Giving them personalized recipes, where user's can save their favourite recipes. Users can add food to the application by typing it in, uploading files, or using the device's camera to take a picture. Users can view what foods they've added to the application.

During my time on this project, I have honed my skills in project planning. Developing and enhancing my programming skills with a wide range of technologies, such as Java, JavaScript, and Python. I have greatly improved my knowledge of web development, which is something before this project I had little exposure to.

2 Poster

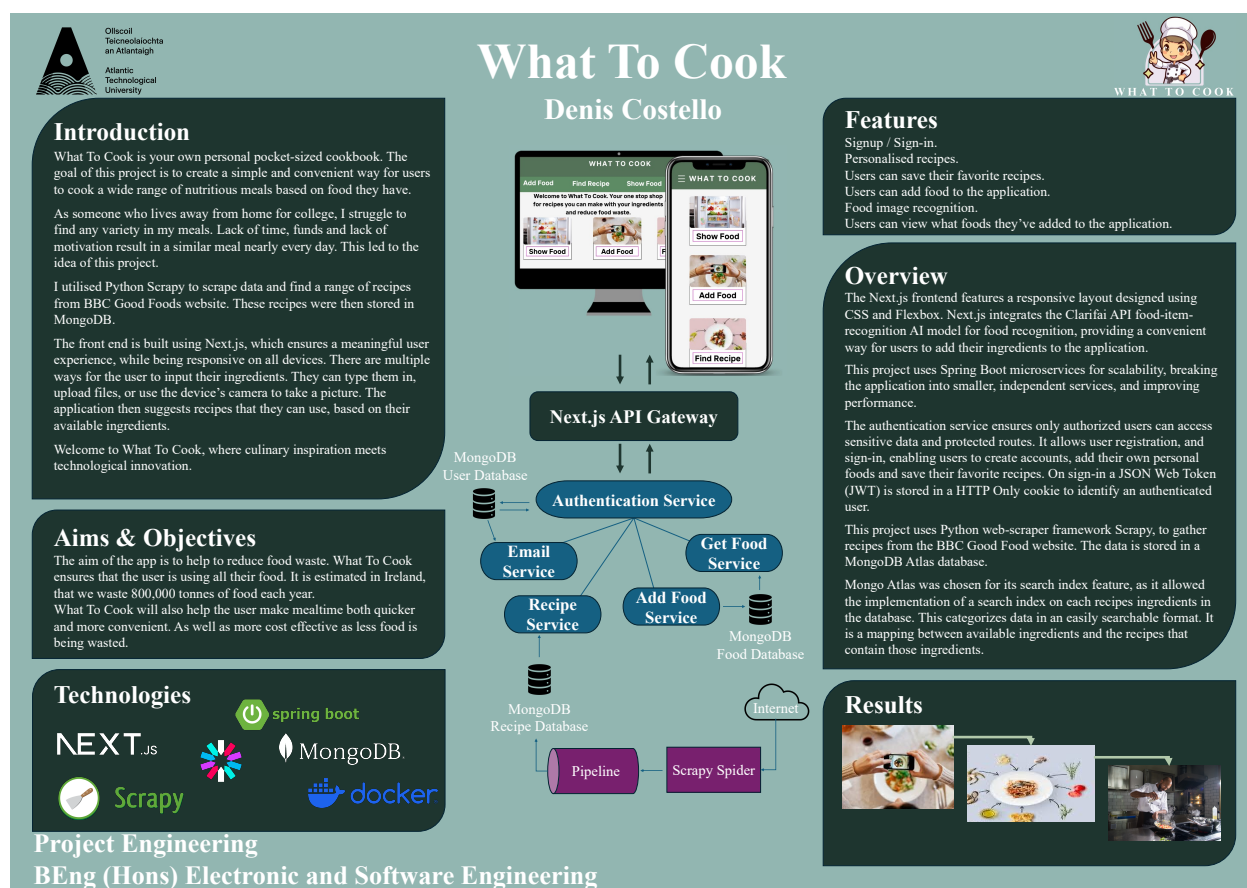


Figure 2-1 – What To Cook Poster

3 Introduction

As someone who lives away from home for college, I struggle to find any variety in my meals. Lack of time, lack of funds and lack of motivation result in a similar meal nearly every day. I wanted to create this project as a solution to create a simple and convenient way for user's to cook a wide range of nutritious meals based on the food they have. The front end is built using Next.js, which ensures a meaningful user experience while being responsive on all devices. There are multiple ways for the user to input their ingredients. They can type them in, upload files, or use the device's camera to take a picture. The application then suggests recipes that they can make, based on their available ingredients.

Given that Progressive Web Apps (PWA) are responsive, fast, and reliable. As well as being accessible through a web browser and compatible with any device, laptop, smartphone, or tablet, I've created What To Cook to be a PWA instead of a native app. Given the discoverability of PWAs and features such as WebRTC getUserMedia/stream API [2] to access the device's camera. It made sense to build a PWA instead of a React Native app for my project.

This project has a microservice architecture using Spring Boot which allowed easier scalability, easy maintenance and creation of REST endpoints, flexibility to add new features, easier to maintain, and increased agility.

This project utilizes Python Scrapy to scrape data and find a range of recipes from the BBC Good Foods website. These recipes were then stored in MongoDB.

Welcome to What To Cook, where culinary inspiration meets technological innovation.

4 Tools and Technology

During the initial stages of this project, I wanted to ensure I chose the right tools and technologies to create the best application I could.

4.1 Microservice Architecture

The microservice architecture paradigm was chosen to build small, containerized Spring Boot applications that are ready to run.

4.2 Spring Boot

Spring Boot is an open-source Java framework used to create microservices at production. Spring Boot makes it easy to create stand-alone, production-grade Spring Applications that you can "just run" [3]. Spring Boot provides easy maintenance and creation of RESTful endpoints.

4.2.1 Spring Boot Authentication

Spring Boot Authentication using Spring Boot security. A web security configuration class has a `OncePerRequestFilter` method [4] that filters for each API gateway request to only allow access to the API gateway with a valid JSON Web Token (JWT).

4.2.2 Spring Cloud OpenFeign

OpenFeign is a declarative HTTP client developed by Netflix. OpenFeign creates a simple process of making HTTP requests [5]. Used with Spring applications and integrates seamlessly with other Spring components.

4.2.3 Spring Boot Email

The java mail sender dependency sends an email to the user's after they sign up. Spring Boot's `JavaMailSender` is an interface in the Spring Framework used for sending email messages in Java applications [6]. The email is sent over a Simple Mail Transfer Protocol (SMTP) server.

4.3 Next.js

Next.js is the react framework for the web [7]. Next.js simplifies the process of building React applications by providing a powerful framework with many built-in features and optimizations.

It provides a way to create React applications with features like server-side rendering, static site generation, automatic code splitting, and easy deployment. It builds on React, things like routing, and state management. Next.js has a lot of features on top of the React framework, that would require multiple 3rd party libraries if it were solely a React app.

4.4 Clarifai API

“Clarifai is the leading Full Stack AI, LLM, and computer vision production platform for modelling unstructured image, video, text, and audio data” [8]. Clarifai API allows developers to integrate AI into their applications.

4.5 MongoDB

MongoDB is a document-oriented database program. MongoDB is a NoSQL database, meaning databases come in a variety of types based on their model. MongoDB provides flexibility and easy scalability.

4.6 MongoDB Atlas

MongoDB Atlas is the cloud version of MongoDB. Atlas automates many administrative tasks and provides additional security and scalability features. Atlas also has an embedded full-text search built on Apache Lucene, Atlas Search eliminates the need to run a separate search system alongside your database [9].

4.7 Scrapy

Scrapy is an open-source web-crawling framework written in Python. Scrapy can extract data from websites in a fast, simple, yet extensible way. Scrapy spiders are classes that define how a certain site will be scraped, including how to perform the crawl and how to extract structured data from their pages.

4.8 Docker

Docker is an open-source platform that allows developers to run their applications in containers. “With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production” [10].

5 Project Architecture

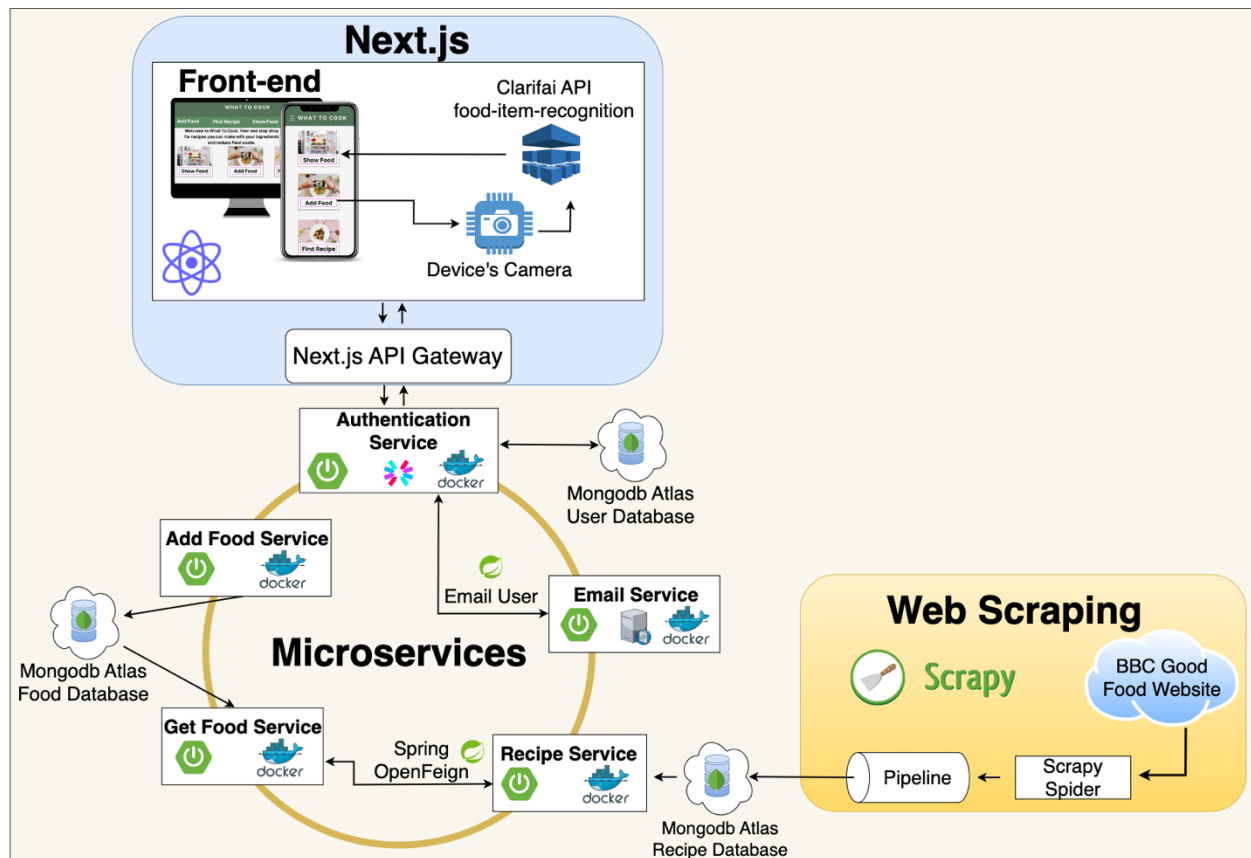


Figure 5-1 Architecture Diagram

6 Project Plan

For my project planning, I used Jira as a project management tool, utilizing the Kanban methodology. Updating the board every week based on tasks due, current tasks I was doing and tasks completed.

Projects / My Kanban Project

KAN board

DC

Epic ▾Label ▾

TO DO 3

Project Video

Extras 29 APR

✓ KAN-68 DC

Project Presentation

Extras 29 APR

✓ KAN-73 DC

Push to kubernetes

Cloud 27 APR

✓ KAN-101

+ Create issue

IN PROGRESS 1

Project Report

Extras 29 APR

✓ KAN-72 DC

DONE 82 ✓

Video of project demo

Documentation 17 DEC 2023

✓ KAN-36 ✓

HttpOnly Cookie

NextJS-Code 03 APR

✓ KAN-99 ✓ DC

Middleware

NextJS-Code 24 APR

✓ KAN-75 ✓ DC

Take snapshot of camera for Clarifai API

NextJS-Code 19 FEB

✓ KAN-62 ✓

Recipe service

Recipe-Creation Coding-java

12 MAR

✓ KAN-83 ✓

7 Project Code

The following section will describe the crucial software elements that allowed me to complete this project. The section will be split into three sections, the front-end which the user interacts with, the backend which is the backbone of the web application, and web scraping for scraping recipes from the internet.

7.1 Front-end

The front-end was built using Next.js. The Next.js application has four vital aspects, the pages folder for routing and API handling, the components folder for UI components, the middleware for protected routes, and the context provider for updating global context and avoiding prop drilling.

7.1.1 Pages folder

The pages folder is used for routing between pages, nested route handling for each recipe ID, and handling API routes to the Spring Boot microservices.

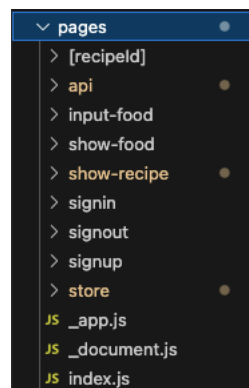


Figure 7-1 – Pages Structure

There are six page routes in the pages folder, '/input-food', '/show-food', '/show-recipe', '/signin', '/signout', and '/signup'. The user can navigate between the pages once logged in.

```

pages > JS _app.js > App
1  import '@styles/globals.css'
2  import {GlobalContextProvider} from './store/globalContext'
3  import Layout from '@components/Layout';
4
5
6  export default function App({ Component, pageProps }) {
7    return (
8      <GlobalContextProvider>
9        <Layout>
10         <Component {...pageProps} />
11       </Layout>
12     </GlobalContextProvider>
13   );
14 }

```

Figure 7-2 - _app.js

The ‘_app.js’ is the entry point for the next.js application, this file acts as a wrapper around the application. Setting up the global context, the layout and the component and its props. The ‘index.js’ or the homepage, is the initial entry point for rendering and is the first screen seen by a logged-in user. The ‘pages/api’ folder is used to define server-side logic. The Next.js API routes are a connection to the Spring Boot RESTful API Endpoints. Handling backend logic, such as data fetching user’s foods, and authenticating the user has a valid JSON Web Token (JWT). The following is a snippet from the ‘sign-in’ in the API folder.

```

pages > api > JS sign-in.js > handler > response
1  import { serialize } from "cookie";
2
3  async function handler(req, res) {
4    const response = await fetch('http://security-service:8080/api/auth/signin', {
5      method: 'POST',
6      body: JSON.stringify(req.body),
7      headers: {
8        'Content-Type': 'application/json'
9      },
10    });
11    const data = await response.json();
12    if (response.ok) {
13      const cookieHeader = response.headers.get('Set-Cookie')
14      const serialised = serialize(cookieHeader)
15      res.setHeader("Set-Cookie", serialised);
16      res.status(200).json(data)
17    }
18    else {
19      res.status(401).json({ message: "Invalid credentials!" })
20    }
21  }
22  export default handler;

```

Figure 7-3 -Sign-in API

This is an asynchronous function, meaning it's a function that operates asynchronously and can execute tasks concurrently or non-blocking. It uses a fetch API to make a POST request to the Spring Boot microservice running on port 8080 with the endpoint `/api/auth/signin`. The POST request sends a body with a JSON object containing the details inputted by the user in the sign-in form. This function awaits a response from the Spring Boot microservice where it checks if the response was successful. Upon receiving a successful response it gets the cookie sent from the response header. Then the cookie is serialized [10] into a format that can be set as a cookie in the client's browser. After serializing the cookie, it's set as the value of the "Set-Cookie" header in the response that will be sent back to the client side. This effectively sets the cookie in the client's browser. If the response is not ok a 401, "unauthorized", is sent as a response which ensures the user will not be signed in.

7.1.2 Components

The components are the main location for the User Interface (UI) components used throughout this project. These components have functionality such as displaying the homepage, allowing the user to add food via the device's camera or uploading a file, displaying the user's food, displaying the top ten recipes suggested for the user, showing the user's favourite recipes, and form layout for signing in and signing out.

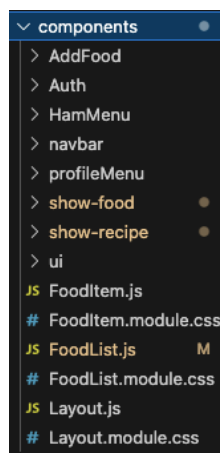


Figure 7-4 – Components Layout

The Add food folder contains the code that utilizes Clarifai API to add via an image. The following code for food recognition is based on a Clarifai tutorial [11] with additional features incorporated and modifications made for my project demands. The camera preview is accessed by pressing the 'Click here to open the camera', which calls the beginCapture callback function.

```
const beginCapture = useCallback(
  async () => {
    if (!cameraPreviewEl.current) {
      return;
    }
    try {
      const stream = await navigator.mediaDevices.getUserMedia({
        video: {
          facingMode: {
            ideal: 'environment'
          }
        }
      });
      setStreamValue(stream);
      cameraPreviewEl.current.srcObject = stream;
      cameraPreviewEl.current.play();
      setCapturing(true);
    } catch (error) {
      console.log('Error accessing camera: ', error);
      setCameraErr('Cannot access the camera')
    }
  },
  [cameraPreviewEl],
);
```

Figure 7-5 – Start Stream

This async callback function initially checks if there is a reference to the video element with cameraPreviewEl.current. Then in a try-catch block, it tries to access the user media device through navigator.mediaDevices.getUserMedia() [2]. It requests to preferably access the outward-facing camera with 'video: { facingMode: { ideal: 'environment' } }'. If the response is successful and access to the device's camera is permitted it sets the stream value to the stream value obtained by the previous line. The reference to the video element is then set to the stream captured (device's camera). The current stream plays within the HTML <video> element, this is how the camera stream is displayed on the webpage. The useState hook, setCapturing is set true which will allow the user to take a snapshot only when the stream is playing. When the user presses the 'Click To Take Picture' button the takeSnapshot callback function is called.

```

const takeSnapshot = useCallback(
  () => {
    if (!cameraPreviewEl.current) {
      return;
    }
    const canvas = document.createElement('canvas');
    canvas.width = 800;
    canvas.height = 600;

    const ctx = canvas.getContext('2d');
    if (!ctx) {
      return;
    }
    ctx.drawImage(cameraPreviewEl.current, 0, 0, canvas.width, canvas.height);
    canvas.toBlob(async (blob) => {
      if (!blob) {
        return null;
      }
      if (snapshot) {
        URL.revokeObjectURL(snapshot)
      }
      setSnapshot(URL.createObjectURL(blob))

      const resp = await postFoodRecognition(blob);
      setResponse(resp);
    });
  },
  []
);

```

Figure 7-6 – Take Snapshot

This callback function ensures there is a reference to a video element, and creates a HTML canvas element [12]. It sets the width to 800 pixels and height to 600 pixels, to match the size of the <video> element. The function draws the video element capturing the camera feed onto a canvas with .drawImage(), effectively to take a 'picture' of what the user's video is currently capturing. The canvas is then converted to a blob with the toBlob() method which creates a Blob object representing the image contained in the canvas. Two if statements then check, one if a blob exists and two if a snapshot is already stored in memory. If a snapshot is already stored in memory it is released from an existing object URL. The setSnapshot state is then set to the latest Blob which is then converted to an object URL which can be rendered in an image. The object URL is then sent to the postFoodRecognition function. The response is set in setResponse.

```

export async function postFoodRecognition(snapshot) {
  const formData = new FormData();
  formData.set('snapshot', snapshot);

  const request = {
    method: 'POST',
    body: formData,
  };

  const resp = await fetch('/api/food-recognition', request);
  const json = await resp.json();
  return json;
}

```

Figure 7-7 – Food Recognition Function

This function receives the Blob object, which represents raw binary data. The Blob object is put into a FormData object for the body of the POST request. The POST request is sent to the `pages/api/food-recognition` using the `fetch` function. Upon receiving a response it is returned to the `AddFood` component.

```

pages > api > JS food-recognition.js > ...
1  import { getSnapshotFileFromRequestBody } from '@utils/getSnapshotFileFromRequestBody';
2  import { predictFood } from '@utils/predictFood';
3
4  export default async function handler(req,res) {
5    if (req.method !== 'POST') {
6      console.warn('Method ${req.method} not allowed for endpoint /food-recognition!');
7      return res.status(405).end();
8    }
9    const file = await getSnapshotFileFromRequestBody(req);
10   const resp = await predictFood(file);
11
12   res.status(200).json(resp);
13 }
14
15 export const config = {
16   api: {
17     bodyParser: false,
18   },
19 };

```

Figure 7-8 – Food Recognition API

This API handler checks if the request method is a POST request, responding with a 405 'Method Not Allowed' if it's not a POST request. Lines 15-19 set the `bodyParser` to false, telling Next.js not to parse the request body. As the body will be parsed by the `busboy` library, `busboy` is used for parsing incoming HTTP requests that contain file uploads [13]. The first `await` calls 'getSnapshotFileFromRequestBody' and passes it the request, this function returns a Buffer that contains all of the image data and is set in a `const` variable `file`. This function then awaits 'predictFood' and passes it to the Buffer 'file'. 'PredictFood' contains the code that connects to

the Clarifai API, it creates grpc metadata, which is a key-value pair, for passing authentication information, as Clarifai only accepts connections through grpc. Once the response comes back, error checking takes place to ensure the data is valid. As the first concept is always the highest value output is set to the first concept. If that concept's value is greater than 0.4, or 40%, the value and name of that food are returned, otherwise, the recognized property is returned as false.

All components have responsive web design, meaning they look good on all devices. This is implemented by using media queries in CSS.

```
/*Phone CSS*/
@media (max-width: 940px) {
  .ResponsiveHeader {
    flex-direction: row;
    align-items: center;
    text-align: center;
  }
  .hamMenu {
    display: flex;
  }
  .links {
    display: none;
  }
  .signout {
    display: none;
  }

  .Title {
    display: flex;
  }

  .NavLinks {
    display: none;
  }
}
```

Figure 7-9 – Navbar CSS

The above code snippet is taken from 'Navbar.module.css'. In this example, a media query has a maximum width breakpoint of 940 pixels. This means any device that is narrower than 940 pixels in will see the hamburger menu and will not see the navigation links under the header.

7.1.3 Middleware

Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly [14]. Middleware is used in this project for protected routes, only allowing verified user's free navigation on the web application.

```
export async function middleware(req, res) {
  const jwt = req.cookies.get('whattocook')?.value
  let verifiedJwt = null
  if(jwt){
    try {
      const response = await fetch('http://security-service:8080/api/auth/getUser', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Cookie': 'whattocook=${jwt}'
        }
      })
      if (response.ok) {
        const data = await response.json();
        verifiedJwt = "valid"
      } else {
        console.log("Not valid")
      }
    } catch (error) {
      console.error('Error validating JWT token:', error);
    }
  }

  if(req.nextUrl.pathname.startsWith('/signin') && !verifiedJwt){
    return
  }
  if(req.nextUrl.pathname.startsWith('/signup') && !verifiedJwt){
    return
  }

  if (!verifiedJwt) {
    return NextResponse.redirect(new URL('/signin', req.url))
  }

  if (req.url.includes('/signin') && verifiedJwt) {
    return NextResponse.redirect(new URL('/', req.url))
  }
}

export const config = {
  matcher: ['/show-food', '/signin', '/signup', '/input-food', '/show-recipe', '/', '/globalContext']
}
```

Figure 7-10 - Middleware

The above snippet is from 'middleware.js', where it initially checks to see if the session cookie 'whattocook' is set in the browser. If there is a session in the browser it sends a POST request to the Spring Boot security microservice to verify the JSON Web Token (JWT), if it's verified the 'verifiedJWT' is set to "valid". Depending on the path of the incoming request, and whether the JWT is verified, the middleware function may redirect the user to different pages. If the path starts with "/signin" or "/signup" and the JWT is not verified, the middleware returns, effectively allowing the request to continue to the sign-in or sign-up pages. If the JWT is not verified and the path doesn't start with "/signin" or "/signup", the middleware redirects the user to the "/signin" page. Allowing a user to sign-in or signup. If the JWT is verified and the path includes "/signin", the middleware redirects the user to the root URL ("/"). So the user does not see the sign-in page if they're in a valid session.

7.1.4 Context API

Context API is used for state management and data sharing without prop drilling. Context API allows the state management access to the context. This means all states managed inside the context provider will update all components listening to the context provider and get the latest data.

```
pages > JS _app.js > App
1  import '@styles/globals.css'
2  import {GlobalContextProvider} from './store/globalContext
3  import Layout from '@components/Layout';
4
5
6  export default function App({ Component, pageProps }) {
7    return (
8      <GlobalContextProvider>
9        <Layout>
10         <Component {...pageProps} />
11       </Layout>
12     </GlobalContextProvider>
13   );
14 }
```

Figure 7-11 Global Context Wrapped Around App

Inside of 'GlobalContextProvider' context provider is wrapped around the entire app so all components in the application have. React hooks such as 'useState', and 'useEffect' are used to create state variables and fetch initial data.

```
export function GlobalContextProvider(props) {
  const [globals, setGlobals] = useState({
    aString: 'init val',
    count: 0,
    hideHamMenu: true,
    hideProfileMenu: true,
    foods: [], user: [],
    recipes: [],
    signInError: [],
    signUpError: [],
    favourites: [],
    dataLoaded: false
  });
}
```

Figure 7-13 Globals
useState.

```
useEffect(() => {
  getAllFoods()
  getAllRecipes()
  getFavRecipes()
}, []);
```

Figure 7-12 Context Provider
useEffect.

The GlobalStateProvider is also used to fetch data such as getting the foods from '/api/getAllFoods', and updating the foods data. Once data is fetched, it is stored in a global state with setGlobals. The 'GlobalContextProvider' returns the value updated back to the application.

```

const context = {
  updateGlobals: editGlobalData,
  theGlobalObject: globals
}

return <GlobalContext.Provider value={context}>
  {props.children}
</GlobalContext.Provider>
}

export default GlobalContext

```

Figure 7-15 Context Provider Return Value

```

async function getAllFoods() {
  const response = await fetch('/api/get-foods', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    }
  });
  let data = await response.json()
  if (response.ok) {
    setGlobals((previousGlobals) => {
      const newGlobals = JSON.parse(JSON.stringify(previousGlobals));
      if (data.foods != null) {
        newGlobals.foods = data.foods;
      }
      newGlobals.dataLoaded = true;
      return newGlobals
    })
  }
}

```

Figure 7-14 Context Provider Get All Foods Function

7.2 Backend

The backend is a microservices architecture with five services, 'Authentication Service', 'Email Service', 'Get Food Service', 'Add Food Service', and 'Recipe Service'.

7.2.1 Authentication Service

The authentication service ensures that only authorized user's can access sensitive data. The following code for authentication service is based on spring documentation [15] and a tutorial I found while researching spring security [16], with additional features incorporated and modifications made for my project demands. This service is spilt into six key packages,

1. Controllers
2. Email
3. Models
4. Payload
5. Repository
6. Security

Controller Package

The controller package handles the API endpoints for incoming requests from the Next.js gateway. It has three primary endpoints: '/api/auth/signin', '/api/auth/signup', and '/api/auth/getUser'.

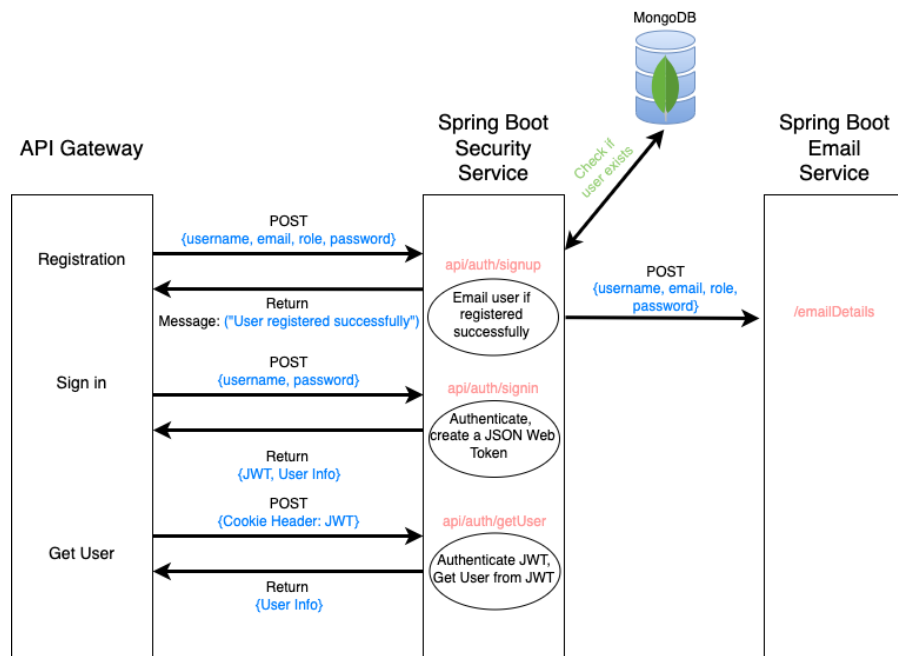


Figure 7-16 – Authentication Service

The above diagram shows the flow of the Next.js API gateway, authentication service, and email service. On a successful login, a JWT is sent back to the Next.js API gateway. There is a utility class to handle the creation and verification of a JWT.

Email Package

The email package contains a public interface called email client, which utilizes openFeign to communicate with the email service running on port 8086. It sends a POST request to the endpoint '/email'.

```

2 usages  Denis Costello
@FeignClient(name="email-service", url = "http://email-service:8086")
public interface EmailClient {
    Denis Costello
    @PostMapping("/email")
    String emailDetails(@RequestBody User user);
}

```

Figure 7-17 – Feign Email Client

Models Package

The model package contains the data structure that will be stored in MongoDB. It has two classes, User, for username, email, password, and role. And Role, for the name and ID of the user to be stored in a role. And an enum Erole, for the role of the user.

```
@Document(collection = "users")
public class User {
    @Id
    private String id;

    3 usages
    @NotBlank
    @Size(max = 20)
    private String username;

    3 usages
    @NotBlank
    @Size(max = 50)
    @Email
    private String email;

    3 usages
    @NotBlank
    @Size(max = 120)
    private String password;

    2 usages
    @DBRef
    private Set<Role> roles = new HashSet<>();
}
```

Figure 7-18 – User Model

Payload Package

This package defines the data taken from and to be sent back to the Next.js API gateway. One example of the payload is, during user registration (/api/auth/signup), if the username already exists a new payload will be returned to the Next.js API gateway.

```
import ie.atu.wtcsecurity.payload.response.MessageResponse;

@PostMapping("/signup")
public ResponseEntity<> registerUser(@Valid @RequestBody SignupRequest signUpRequest) {
    if (userRepository.existsByUsername(signUpRequest.getUsername())) {
        return ResponseEntity
            .badRequest()
            .body(new MessageResponse("Error: Username is already taken!"));
    }
}
```

Figure 7-19 – Message Response Payload

Repository Package

The repository package contains the interface to the Mongo Atlas repository. Accessing and saving data to the Mongo repository.

Security Package

The security package manages authentication and authorization. Inside the security package, there is the WebSecurityConfig class, which is the heart of the service, as it configures the security for the application. Using `@EnableMethodSecurity` [17] annotation to enable method-level security globally, to authenticate and authorize the API endpoints. Thus, ensuring that only authorized user's get access to the endpoints.

```
@EnableMethodSecurity
public class WebSecurityConfig {
```

Figure 7-20 – Enable Security

This class has a method `AuthTokenFilter`, which is a filter that intercepts incoming requests and validates JWT tokens. If the user is valid, it sets the user's authentication context. This class also has an instance of the Spring Security `DaoAuthenticationProvider` [18], to authenticate the user by using `UserDetailsService` and `PasswordEncoder`.

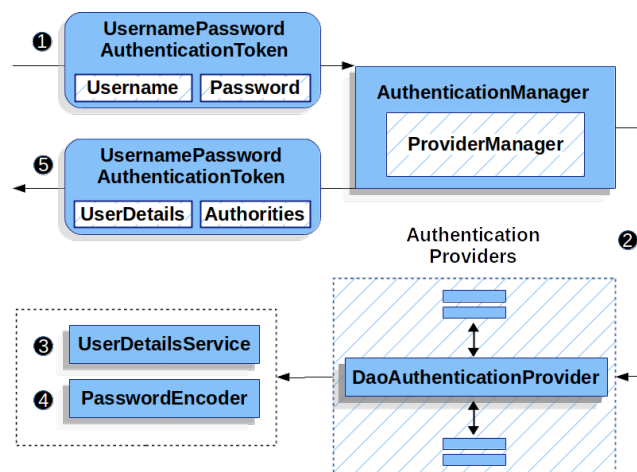


Figure 7-21 – DaoAuthenticationProvider
(Taken from Spring documentation [18])

7.2.2 Add Food Service

This microservice has two primary endpoints, `/food/addNewFood` and `/food/addNewRecipeId`. The first endpoint takes in an instance of the `InputFood` class and passes it to the `addFood` method `inputFoodService` class.

```
1 usage  Denis Costello
public InputFood addFood(InputFood inputFood) {
    Optional<InputFood> existingFood = foodRepository.findById(inputFood.getUserId());

    if (existingFood.isPresent()) {
        InputFood food = existingFood.get();
        List<String> foodList = food.getFoods();
        if (foodList != null) {
            foodList.add(inputFood.getFoodName());
            food.setFoods(foodList);
        } else {
            List<String> newfoodList = new ArrayList<>();
            newfoodList.add(inputFood.getFoodName());
            food.setFoods(newfoodList);
        }
        return foodRepository.save(food);
    }
    return inputFood;
}
```

Figure 7-22 – addFood in input food service.

The `addFood` method checks to see if the user's ID is already in the database. If a user exists it checks to see if a user has added food already and has a list of their foods. If the food list is not null it adds the new food to the existing foodlist. However, if the user has no food list it creates a new `ArrayList`, adds the food to the new list, and sets the foods to the new user's food. The list is then saved in the MongoDB Atlas database.

The `addrecipeId` method has a very similar logic to the `addFood` method. This service runs on port 8082 and connects to the food database.

7.2.3 Get Food Service

This microservice has three key endpoints, `/food/getAllFood`, `food/getFoodForRecipe`, and `food/getFavRecipes`. The `getAllFood` endpoint is used to return a `ResponseEntity` hashmap

with a key-value pair of a string called foods and a list of strings for the food names for the user's that request this endpoint. The other two endpoints have similar functionality.

7.2.4 Recipe Service

This microservice gets called from the get food microservice by using openFeign. The get food service sends a POST request with a body of a list of the foods the user has. The list of foods is then passed to a repository implementation.

```
public List<FoodData> findByText(List<String> text) {
    List<FoodData> recipes = new ArrayList<>();

    MongoDBDatabase database = client.getDatabase("ScrapedRecipes");
    MongoClientDatabase collection = database.getCollection("recipes");

    AggregateIterable<Document> result = collection.aggregate(Arrays.asList(new Document("$search",
        new Document("index", "default")
            .append("text",
                new Document("query", text)
                    .append("path", "ingredients"))),
        new Document("$project",
            new Document("$meta", "searchScore")
                .append("title", 1)
                .append("url", 1)
                .append("description", 1)
                .append("prepTime", 1)
                .append("cookTime", 1)
                .append("ingredients", 1)
                .append("recipeId", 1)
                .append("image", 1)
                .append("steps", 1)),
            new Document("$sort",
                new Document("$meta", "textScore"))));

    //result.forEach(doc -> recipes.add(converter.read(FoodData.class, doc)));

    //only add the first (highest textScore) to recipes
    Iterator<Document> iterator = result.iterator();
    for (int i = 0; i < 10 && iterator.hasNext(); i++) {
        recipes.add(converter.read(FoodData.class, iterator.next()));
    }

    return recipes;
}
```

Figure 7-23 – Search Repository Implementation

The following code is from Mongo Atlas aggregation 'export to language' feature. I used the Mongo Atlas aggregation pipeline to create a search index on my recipes scraped database.

The screenshot shows the Mongo Atlas aggregation pipeline configuration and output. The top section displays a preview of documents from the 'recipes' collection. The bottom section shows the aggregation pipeline configuration for the 'Search' stage, which includes a search index and a text search query. The output of the pipeline is shown as a sample of 10 documents.

Preview of documents

- Document 1: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 3, title: "Quick tomato soup with cheesy garlic"
- Document 2: description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 4, title: "Christmas dinner pie with roasted"
- Document 3: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 8, title: "One-tray roast chic"
- Document 4: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 11, title: "One-tray roast chic"

Stage 1: Search

Output after Search stage (Sample of 10 documents)

- Document 1: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 3, title: "Quick tomato soup with cheesy garlic"
- Document 2: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 4, title: "Christmas dinner pie with roasted"
- Document 3: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 8, title: "One-tray roast chic"
- Document 4: url: "https://www.bbcgoodfood.com/recipes/qu...tomato-soup-cheesy-garlic-dl...", description: "Cook a special Christmas Day meal for one with this pie. It's filled w...", recipeId: 11, title: "One-tray roast chic"

Figure 7-24 – Mongo Atlas Aggregation Pipeline Search Index

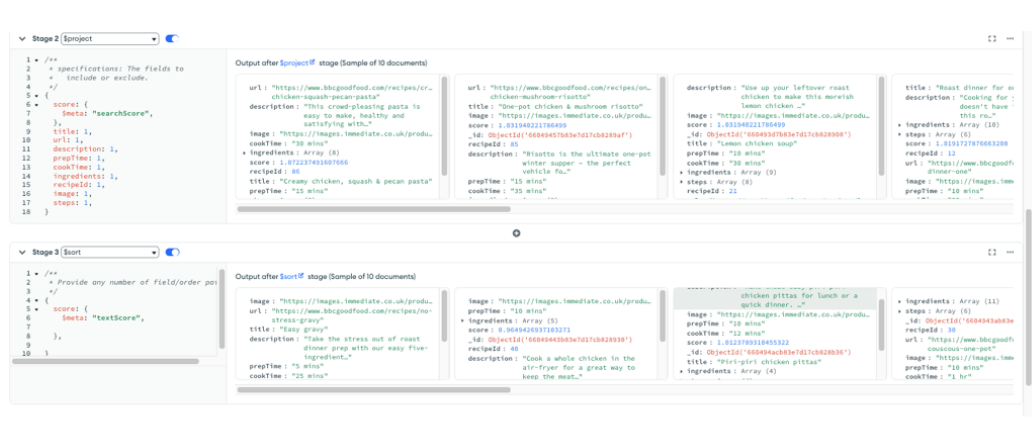


Figure 7-25 - Mongo Atlas Aggregation Pipeline Project and Sort

Above are two snippets where I used Mongo Atlas' search index visual editor, to incorporate a search index on the ingredients in each recipe of the database. Once I got the search index working how I wanted I used the export to language feature and selected Java.

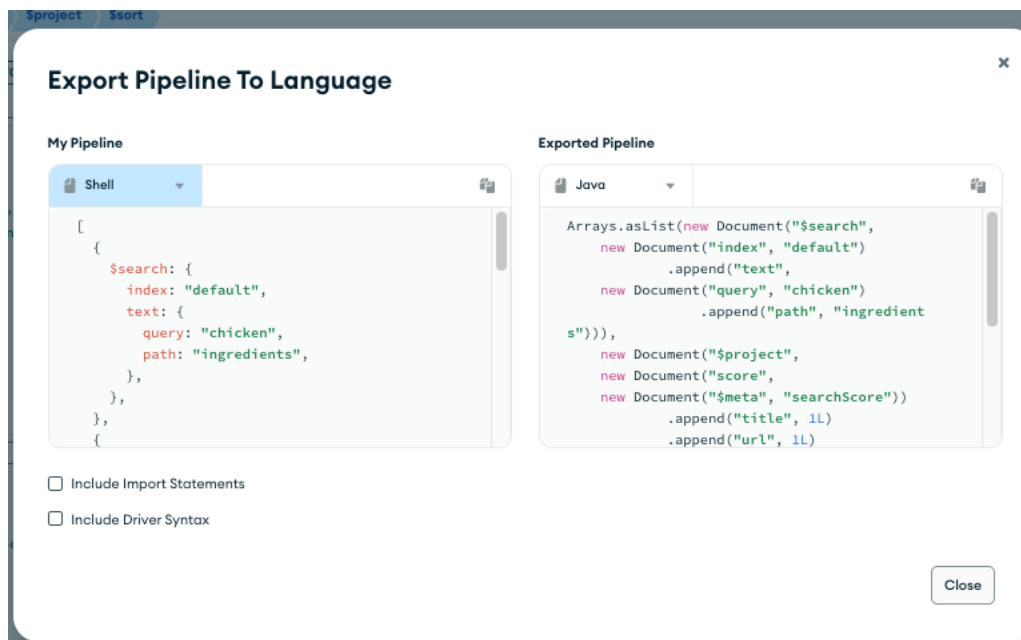


Figure 7-26 – Mongo Atlas Export To Language

7.2.5 Email Service

The email service is invoked by the authentication service with openFeign, as previously stated above. It has an endpoint of '/email' where it takes in an instance of the user class. Getting the username and email from the instance it is then passed to the sendEmail method in the email service.

```
@Service
public class EmailSendService {
    @Autowired
    private JavaMailSender mailSender;

    1 usage   Denis Costello
    public void sendEmail(String toEmail, String subject, String body){
        SimpleMailMessage message= new SimpleMailMessage();
        message.setFrom("whattocookteam@gmail.com");
        message.setTo(toEmail);
        message.setText(body);
        message.setSubject(subject);

        mailSender.send(message);
        System.out.println("Mail sent successfully... ");
    }
}
```

Figure 7-27 – Send Email Method

Above is a snippet of the sendEmail method. Where the Spring Boot's JavaMailSender is injected for sending an email. The email will be sent from an email account I have created, whattocookteam@gmail.com. It will be sent to the email address the user has provided in the signup form. The body and the subject are set to the data passed from the controller. The email is sent over a Simple Mail Transfer Protocol (SMTP) server. The SMTP server properties are set in the application.properties file.

```
spring.config.import=file:env.properties
spring.application.name=WTC-EmailService
server.port=8086
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=${mailUsername}
spring.mail.password=${mailPassword}
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

Figure 7-28 - SMTP Server Properties

7.3 Web Scraping

This project uses Python web-scraper framework Scrapy [19], to gather recipes from the BBC Good Food website. The data is stored in a MongoDB Atlas database. When researching Scrapy I found a very helpful tutorial on web scraping with Scrapy [20]. This tutorial scraped books from <https://books.toscrape.com/>, a website to practice web scraping. I used to knowledge gained from this tutorial to create a Scrapy spider and use it on the BBC Good Food website [21] and scrape recipes. All the data was scraped and put in a Scrapy item. Scrapy items allowed me to structure my data so it would be in JSON format. Then using Scrapy Pipelines and PyMongo [22] I was able to save the data to Mongo Atlas.

```
from itemadapter import ItemAdapter

class FoodscraperPipeline:
    def process_item(self, item, spider):
        return item

import pymongo

class SaveToMongoPipeline:
    def __init__(self):
        self.mongo_uri = 'mongodb+srv://root:root@fypcluster.zdqrt70.mongodb.net/?retryWrites=true&majority6appName=FYPCluster'
        self.mongo_db = 'ScrapedRecipes'
        self.client = pymongo.MongoClient(self.mongo_uri)
        self.db = self.client[self.mongo_db]

    def process_item(self, item, spider):
        food_data = {
            "recipeId": item["recipeId"],
            "url": item["url"],
            "title": item["title"],
            "description": item["description"],
            "image": item["image"],
            "prepTime": item["prepTime"],
            "cookTime": item["cookTime"],
            "ingredients": item["ingredients"],
            "steps": item["steps"],
        }

        self.db.foods.insert_one(food_data)

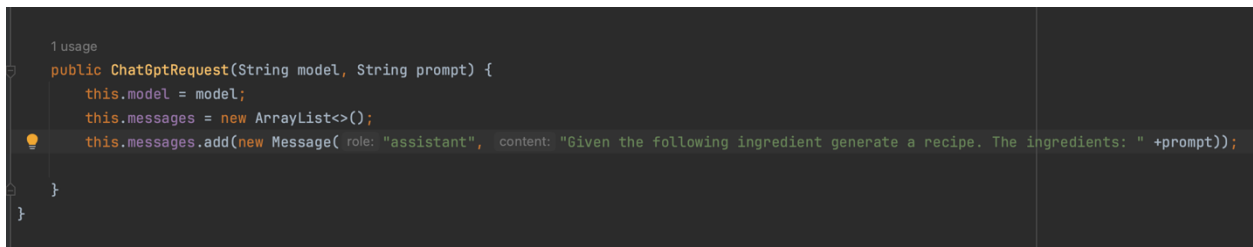
        return item

    def close_spider(self, spider):
        self.client.close()
```

Figure 7-29 - Saving Items To Mongo Atlas

8 Issues

This project was certainly not smooth sailing. I have faced many issues throughout this project which led to me incorporating workarounds. One key aspect where I needed to implement a workaround is the recipe generation. I initially tried to create the recipes with Openai's GPT API [23] integrated with my spring boot application.



```
1 usage
public ChatGptRequest(String model, String prompt) {
    this.model = model;
    this.messages = new ArrayList<>();
    this.messages.add(new Message( role: "assistant", content: "Given the following ingredient generate a recipe. The ingredients: " +prompt));
}
}
```

Figure 8-1 ChatGPT Prompt

The above snippet of code is a prompt for the OpenAI's API. Where 'prompt' is the query passed in by the user, i.e. the user's foods. This prompt was successful and would return a quality recipe. However, when I altered the prompt asking to create 10 recipes the quality of the response was not good. This led to me creating a workaround and opting to use the Python web scrapper to generate recipes instead. N.B. Upon further discussion with my supervisor Brian O'Shea, he told me the prompting method that was used was not sufficient. The correct way to prompt for recipes would be to prompt the API for 10 recipe titles, and then for each of those titles send a new prompt for the cooking methods of each of those titles. However, at the time of this discovery, it was too late to implement the changes.

Problems also happened when web scraping with Scrapy. The data is stored in a Mongo Atlas database. However, for the Atlas free account, a user is only permitted 512MB of storage. For the initial spider that was ran, it took up 381MB of storage, nearly 75% of the allocated storage space. This issue would be salvaged by deleting the data in the database and running the spider again. However this time the spider was capped at how many pages it could scrape.

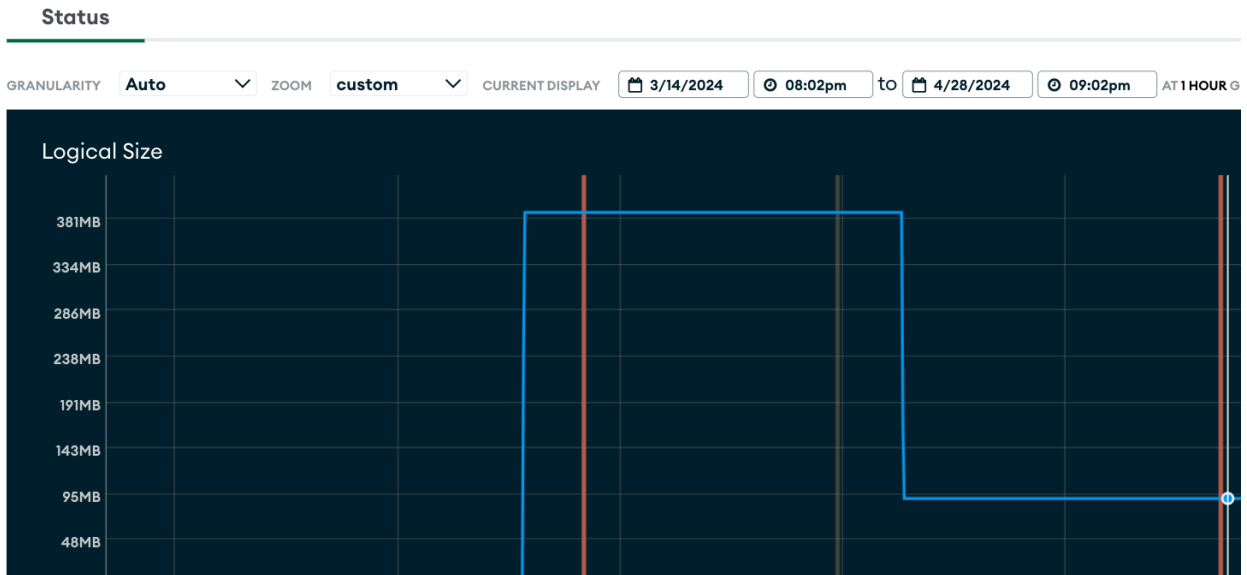


Figure 8-2 MongoDB Atlas Storage

Another issue that I encountered during the later stages of this project was hosting the application. After dockerizing my application, the next step in my project plan was to host the docker images on Amazon Elastic Container Service (ECS) clusters with an Amazon ECR repository. However, after working alongside my supervisor Brian O'Shea we realized this feature is not available for the student plan I am on. This pushed my project completion date back. I attempted to host the images on a Kubernetes cluster but with other project deliverables due, such as poster, report, video, and presentation. It did not have the leverage in the project plan to find a solution for the following issue.

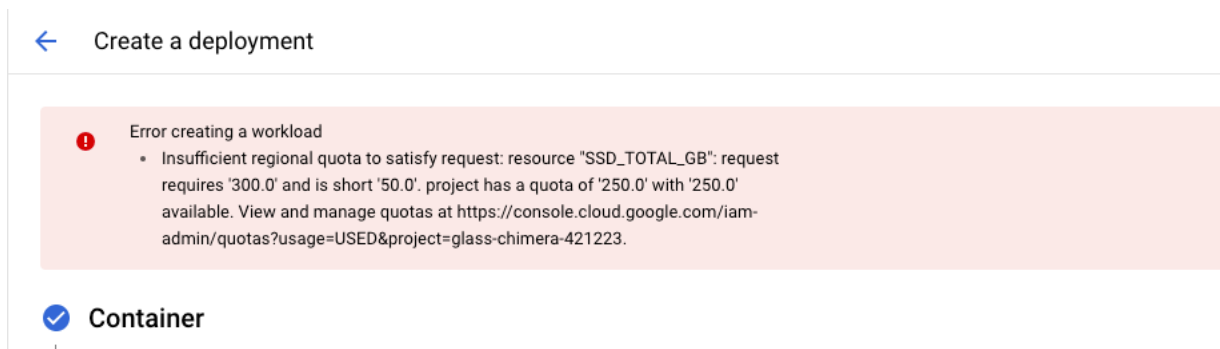


Figure 8-3 Kubernetes Workload Error

9 Teamwork

Engaging and collaborating with my peers during my time working on this project benefited me enormously. From working on project deliverables such as the project poster and presentation, getting feedback was very beneficial as they spotted many mistakes that I did not. Also just communicating with students near me in project labs about blockers in my project helped me overcome issues as many times they would look at the problem a different way from me.

10 Ethics

While developing What To Cook, several ethical considerations must permit attention. One key concern is the use of the camera on the user's device's camera to take images of their food. The user has the option to stop the camera stream or even turn off the device's camera in the device's browser settings, the user's data security comes to the forefront of this project.

Another ethical consideration for this project is web scraping. The aspect of copyright privileges must come into consideration as the data is being taken from another website. The recipe data is not mine and I do not take any credit for creating the recipes. I have included a link to the exact recipe on the BBC Good Food website for each respective recipe.

11 Conclusion

Throughout this project, I've learnt a great deal about the development of a full-stack application. I set out the goal of this project to reduce food waste by enabling the users to make interesting recipes with their ingredients and I believe I have achieved that.

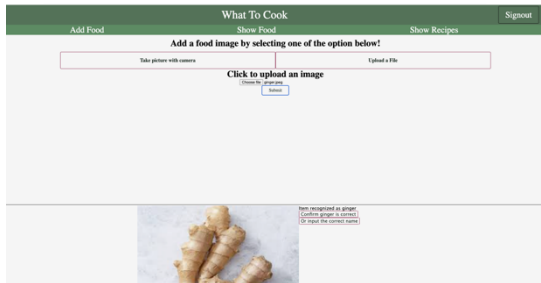


Figure 11-1 Add Food By Upload



Figure 11-2 Add Food By Camera



Figure 11-3 View Users Food



Figure 11-4 Recipe For User

As seen above the user can input their food via upload or the device's camera. The user can view the foods they've added and their favourite recipes. The user can also view what recipes they can make from the foods stored.

I believe this project has great potential for future development and given the structure I have created for this project, i.e. microservice architecture, it would be seamless.

12 References

- [1] Irish Examiner, "Irish Examiner," Irish Examiner, 29 June 2023. [Online]. Available: <https://www.irishexaminer.com/news/arid-41172826.html>. [Accessed 20 April 2024].
- [2] mdn web docs, "mdn web docs," mdn web docs, 28 December 2023. [Online]. Available: WebRTC getUserMedia/stream API . [Accessed 20 April 2024].
- [3] Spring, "Spring Boot," Spring, 21 March 2024. [Online]. Available: <https://spring.io/projects/spring-boot>. [Accessed 19 April 2024].
- [4] Spring, "Spring docs," Spring, 12 June 2003. [Online]. Available: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.web.filter.OncePerRequestFilter.html>. [Accessed 19 April 2024].
- [5] P. Lennon, *Microservice communication & Rest Principals*, Galway: Lecturer, ATU Galway, 2024.
- [6] Spring, "Spring Email," Spring, 10 June 2024. [Online]. Available: <https://docs.spring.io/spring-framework/reference/integration/email.html#>. [Accessed 23 April 2024].
- [7] Vercel, "About React and Next.js," Vercel, 12 April 2024. [Online]. Available: <https://nextjs.org/learn/react-foundations/what-is-react-and-nextjs>. [Accessed 23 April 2024].
- [8] Clarifai, "Clarifai," Clarifai, 19 February 2024. [Online]. Available: <https://www.clarifai.com/>. [Accessed 20 April 2024].
- [9] MongoDB, "What is MongoDB Atlas Search," MongoDB, 10 October 2023. [Online]. Available: <https://www.mongodb.com/docs/Atlas/Atlas->

search/#:~:text=Atlas%20Search%20is%20an%20embedded,search%20system%20alongside%20your%20database.. [Accessed 20 April 2024].

[1] npmjs, "npmjs cookie," npmjs, 10 November 2023. [Online]. Available:

0] <https://www.npmjs.com/package/cookie>. [Accessed 23 April 2024].

[1] Clarifai, "Add AI to a Node.js Web App," Clarifai, 10 June 2023. [Online]. Available:

1] <https://docs.clarifai.com/tutorials/node-js-tutorial/>. [Accessed 10 January 2024].

[1] mdn web docs, "<canvas>: The Graphics Canvas element," mdn web docs, 22 February

2] 2024. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas>. [Accessed 22 April 2024].

[1] npm, "npm busboy," npm, 11 March 2022. [Online]. Available:

3] <https://www.npmjs.com/package/busboy>. [Accessed 20 April 2024].

[1] Vercel, "Middleware," Vercel, 20 June 2024. [Online]. Available:

4] <https://nextjs.org/docs/app/building-your-application/routing/middleware>. [Accessed 25 April 2024].

[1] Spring, "Securing a Web Application," Spring, 10 April 2024. [Online]. Available:

5] <https://spring.io/guides/gs/securing-web>. [Accessed 26 April 2024].

[1] bezkoder, "Spring Boot Token based Authentication with Spring Security & JWT," bezkoder,

6] 25 January 2024. [Online]. Available: <https://www.bezkoder.com/spring-boot-jwt-authentication/>. [Accessed 10 March 2024].

[1] Baeldung, "Spring @EnableMethodSecurity Annotation," Baeldung, 15 January 2022.

7] [Online]. Available: <https://www.baeldung.com/spring-enablemethodsecurity>. [Accessed 14 March 2024].

[1] Spring, "DaoAuthenticationProvider," Spring, 10 April 2024. [Online]. [Accessed 15 April

8] 2024].

[1] Scrapy, "Scrapy," Scrapy, 20 April 2024. [Online]. Available: <https://Scrapy.org/>. [Accessed 9] 11 March 2024].

[2] freecodecamp, "YouTube - Scrapy Course – Python Web Scraping for Beginners,"
0] freecodecamp, 23 April 2023. [Online]. Available:
https://www.youtube.com/watch?v=mBoX_JCKZTE&t=2063s. [Accessed 20 March 2024].

[2] BBC Good Food, "BBC Good Food," BBC, 21 April 2024. [Online]. Available:
1] <https://www.bbcgoodfood.com/>. [Accessed 21 April 2024].

[2] PyMongo, "PyMongo 4.7.0 Documentation," PyMongo, 20 April 2024. [Online]. Available:
2] <https://pymongo.readthedocs.io/en/stable/>. [Accessed 27 April 2024].

[2] Openai, "openai," Openai, 11 March 2024. [Online]. Available:
3] <https://platform.openai.com/>. [Accessed 26 April 2024].