Computer Science 499 – Microservices

Design Specification

Project By:
Jeremiah Lynn

Mentored By:
Balasubramanian Kandaswamy

Table of Contents

Project Overview

The project will be a simple, recipe sharing social media platform. I've identified the main components required and isolated them into various services. The goal in deciding these decision boundaries was to minimize interservice communication. This will allow for low coupling, increased scalability, and independent deployment. Wherever boundary crossing is required, interfaces must be strictly enforced to maintain said independent deployment.

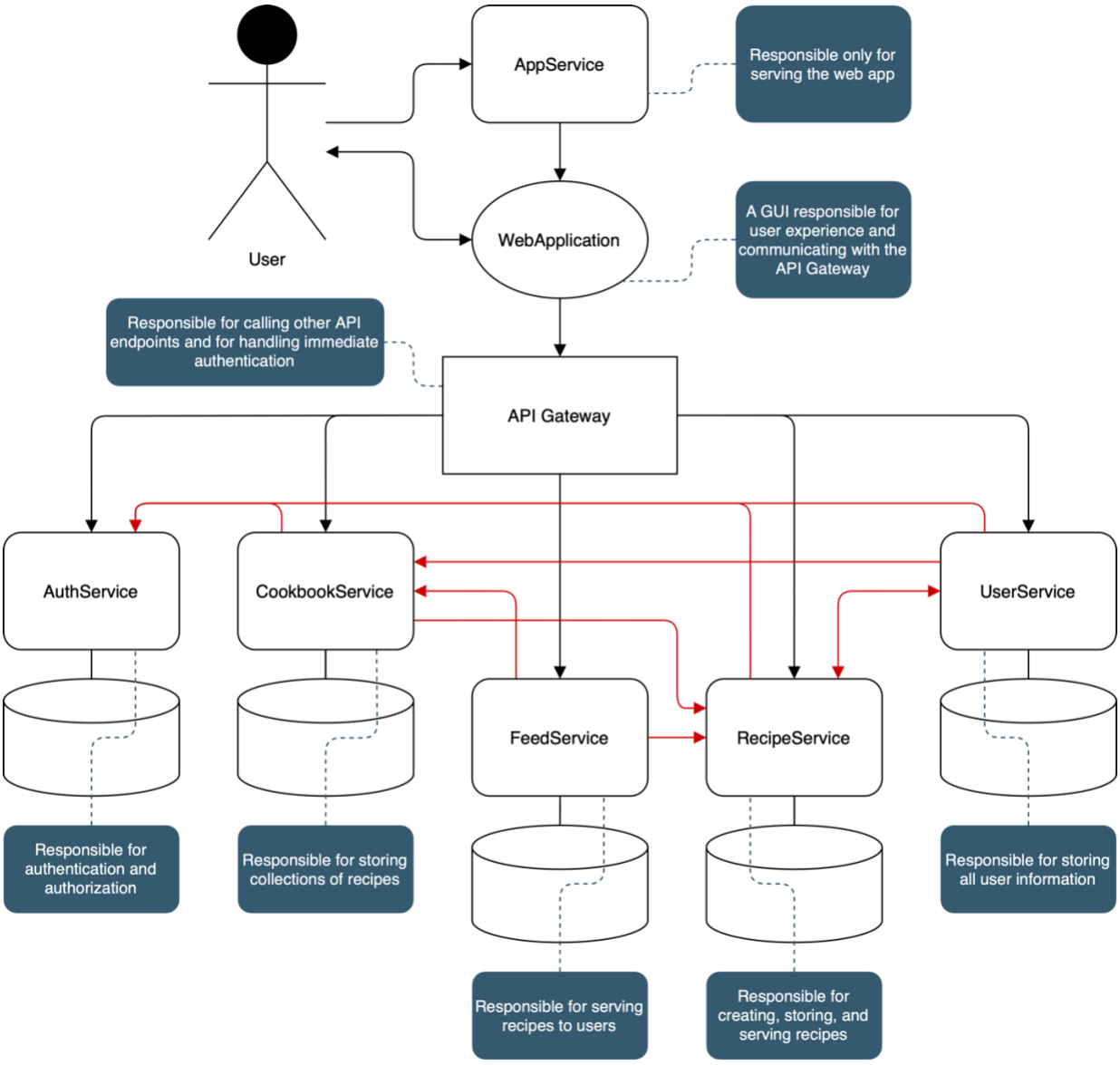

Figure 1. Recipe sharing social media overview.

AppService

The AppService is responsible strictly for serving the web application. More information can be found about the web app in the next section.

API Endpoints

*	A single endpoint used to get the application. All routing will be handled be the application itself.

WebApplication

The WebApplication will be a single page application (SPA) developed using the React library. The application itself will serve as a graphical wrapper for all our API endpoints. This is the intended way the application should be run, while technically one could use curl requests to navigate and manage the app.

Pages

/register	A page used for registering a new account
/login	A page used for logging in to an existing account
/cookbook/view?id=ID	A page used to view a cookbook given a certain ID
/cookbook/create	A page used to create a new cookbook
/home	A page used to view the home feed, a feed of recipes catered to the users previously liked recipes
/search	A page used to query recipes
/recipe/view?id=ID	A page used to view a complete recipe

/recipe/create	A page used to create a new recipe
/recipe/edit?id=ID	A page used to edit a recipe that is owned by the currently logged in user
/user	A page to view a user's home page

API Dependencies

API Gateway	The application will send all the requests to the API Gateway, who is responsible for sending them elsewhere.

API Gateway

The API Gateway is the external connection point to the services. It will be the only public facing connection, and all other services will be internally facing only. The goal of the gateway is to simplify requests by providing a single place to make all requests. This requires special care as it will be a single point of failure for our application and can be a choke point for all requests. Therefore, minimum processes should occur on the gateway, and as much processing as possible should be delegated the services it redirects to.

Additionally, the API Gateway will be responsible for the first layer of authentication. Every API will be able to accept some sort of access token. From the WebApplication, this will be a in the form of a cookie. Otherwise, this will just be a bearer access token on an individual request. For cookie-based authentication, we will convert the cookie to a bearer access token before sending the routing the request. This way, all the individual services need not be concerned about handling cookies.

API Routes

/auth/*	Routes all authentication request to the AuthService
/cookbook/*	Routes all cookbook requests to the CookbookService
/feed/*	Routes all feed requests to the FeedService

/recipe/*	Routes all recipe requests to the RecipeService
/user/*	Routes all user requests to the UserService

API Dependencies

The API Gateway is responsible for routing all requests to the correct services. Therefore, the gateway itself is dependent on each service and their endpoints.

AuthService
CookbookService
FeedService
RecipeService
UserService

AuthService

The AuthService is primarily responsible for authenticating users and provide authorization mechanisms to be used in other services. When a user authenticates, both a cookie is set, and an access token is returned. These values are the same, and they allow use from both the browser and any HTTP request utility. As noted in the API Gateway description, either the access token needs to be sent as a header, or a cookie with the access token needs to be sent.

Certain actions or views may require that the user is properly authorized. This will be handled as role-based authorization. The proper handling of authorization will be twofold. First the correct role needs to be set in the AuthService. Secondly, any time an authorization check is required, the service requiring authorization for an action should contact the AuthService to receive a role for a user, then the service itself is responsible for properly applying the role when deciding whether the action should or should not be permitted.

API Endpoints

(public) POST /register	Registers a new user.
	
	Request {
	  Body {
	  username: string & unique
	  password: string
	  }
	}
	
	Response {
	  Body {
	  access_token: “abcdefg…”
	  }
	  Set-Cookie: access-token=abcdefg…
	}

(public) POST /login	Logs in an existing user.
	
	Request {
	  Body {
	  email: string & unique
	  username: string & unique
	  password: string
	  }
	}
	
	Response {
	  Body {
	  access_token: “abcdefg…”
	  }
	  Set-Cookie: access-token=abcdefg…
	}

(public) GET /logout	Useful only in the WebApplication, removes the access token cookie.
	
	Response {
	  Set-Cookie: access-token=deleted
	}

(public) GET /role	Used to access a user's role data.
	
	Request {
	  Body {
	  id: string
	  }
	}
	
	Response {
	  Body {
	  role: IRole
	  }
	}

(public) PUT /role	Used to set a user's role data.
	
	Request {
	  Body {
	  id: string
	  role: IRole
	  }
	}

(private) DELETE /	Used to delete a user's authentication data.
	
	Request {
	  Body {
	  id: string
	  }
	}

Interfaces

IRole	Bitflag {
	  0b00000001 // Can see private posts.
	  0b00000010 // Can see private cookbooks.
	  0b00000100 // Can see user roles.
	  0b00001000 // Can delete posts.
	  0b00010000 // Can delete cookbooks.
	  0b00100000 // Can adjust user roles.
	  0b01000000 // Can suspend users.
	  0b10000000 // Can delete users.
	}

CookbookService

The CookbookService handles all cookbook related data, which represents a saved collection of recipes. These collections can be organized by sections but are otherwise they just reference recipes from the RecipeService. Cookbooks themselves can either be set to private, unlisted, or public.

Public cookbooks can be discovered through searching, unlisted cookbooks can only be viewed by exact URL reference, and private cookbooks are only viewable by their owners.

Public or unlisted cookbooks can be saved or subscribed to. If you save a cookbook, a copy will be made in your cookbook collection, and you can view, edit, or delete it from there. Saved cookbooks are restricted to unlisted or private. If you subscribe to a cookbook, it will not copy the cookbook, but instead be a mirror of the current state of the cookbook you subscribed to. This means you will see changes they have made, but it also means that if the cookbook is made private or deleted, you will lose access.

Recipes added to a cookbook can include their version number. This way you can be sure the recipe will always remain in your cookbook, even if the recipe version or recipe itself is deleted. If no version number is specified, the latest public version will be served, but if there is no latest public version, or if the recipe itself has been deleted, then the saved recipe will be lost.

API Endpoints

(public) GET /	Used to get a cookbook.
	
	Request {
	  Body {
	  id: string
	  }
	}
	
	Response {
	  Body {
	  cookbook: ICookbook
	  }
	}

(public) DELETE /	Used to delete a cookbook.
	
	Request {
	  Body {
	  id: string
	  }
	}
	
	This will require updating all recipes reference counts in the RecipeService.
	
	This can require checking authorization in the AuthService

(public) PATCH /	Used to update a cookbook.
	
	Request {
	  Body {
	  id: string
	  visibility: ‘private’ | ‘public’ | ‘unlisted’
	  recipes: ICookbookSection[]
	  }
	}
	
	This will require updating all recipes reference counts in the RecipeService

(private) POST /copy	Used to make a copy of a cookbook, used in the save functionality. The id returned is the id of the new recipe.
	
	Request {
	  Body {
	  id: string
	  new_owner: string
	  }
	}
	
	Response {
	  Body {
	  id: string
	  }
	}
	
	This will require updating all recipes reference counts in the RecipeService

Interfaces

ICookbook	{
	owner: string
	times_saved: number
	subscriptions: number
	is_a_copy: boolean
	visibility: 'private' | 'public' | 'unlisted'
	recipes: ICookbookSection[]
	}

ICookbookSection	{
	title: string
	recipes: ICookbookRecipe[]
	}

ICookbookRecipe	{
	id: string
	version: string | null
	}

API Dependencies

RecipeService	Cookbooks will not contain any recipes or recipe references themselves, but rather recipe ids. This way we need not cross the boundary for serving cookbooks. However, recipes have some specific reference counting techniques and therefore whenever we reference a recipe in a cookbook, the recipe needs to be informed.
AuthService	Most checks should be completable by checking the user id in the JWT, but if a user tries to access to complete an action not generally allowed (with nonmatching owner id), then the AuthService will be required for getting a user's role data before deciding whether the user should be permitted to do the action.

FeedService

The FeedService is responsible for serving scrollable collections to the user. It is also responsible for all searching functionality.

There will be a heavy reliance on the RecipeService, as the FeedService itself will only be maintaining basic information on a user's preferences but will be using the recipe service for the actual querying and returning recipes to the user.

API Endpoints

(public) GET /home	Used to get a user's home feed, or a generic one if no user is logged in. Works in batches.
	
	Request {
	  Body {
	  items: number
	  set: number
	  }
	}
	
	Response {
	  Body {
	  recipes: IRecipe[]
	  }
	}
	
	This will require getting a recipe collection from the RecipeService

(public) GET /query	Used to get a feed based on a query.
	
	Request {
	Body {
	type: ‘cookbook’ | ‘recipe’
	items: number
	set: number
	query: IFeedQuery
	}
	}
	
	Response {
	Body {
	recipes: IRecipe[] | null
	cookbooks: ICookbook[] | null
	}
	}
	
	This can require getting a recipe collection from the RecipeService.
	
	This will require getting a cookbook collection from the CookbookService.

Interfaces

IFeedQuery	{
	title: string
	// to-do
	}

API Dependencies

RecipeService	Used to get and return a collection of recipes to the user.
CookbookService	Used to get and return a collection of cookbooks to the user.

RecipeService

The RecipeService will handle all recipe related data.

Recipes themselves should be version controlled. Each version of a recipe is no longer editable, when edits are made a new version must be submitted. This will protect cookbooks who rely on older versions of a recipe. Recipe versions can be deleted, but they are not deleted immediately. Rather every time a cookbook adds a recipe, a count on the recipe itself is incremented, and when a recipe is removed, the count is decremented. If a deleted recipe's count reaches 0, the recipe will be completely removed.

Recipes and recipe versions can each have visibility set to public, private, unlisted, or deleted (in case of deleted). Public recipes are discoverable through search, unlisted through URL only (except for the user who made the recipe, who will see it in their recipes), private cannot be seen by anyone, and deleted will function similarly as unlisted, except the user that made the recipe won't be able to see it except through URL. Between the recipe and the recipe version, the more restrictive visibility of the two will be considered.

All recipes will maintain a latest public version, and this will be the version served if no version is specified upon the request.

API Endpoints

(public) GET /	Used to get a recipe by id.
	
	Request {
	  Body {
	  id: string
	  version: string | null
	  }
	}
	
	Response {
	  Body {
	  recipe: IRecipe
	  }
	}

(public) POST /	Used to create a new recipe.
	
	Request {
	  Body {
	  visibility: ‘public’ | ‘private’ | ‘unlisted’
	  data: IRecipeData
	  }
	}
	
	This will require updating a collection in the UserService and checking for user timeout.

(public) PATCH /	Used to edit an existing recipe, will create a new version if data field is included.
	
	Request {
	  Body {
	  id: string
	  version: string | null
	  visibility: ‘public’ | ‘private’ | ‘unlisted’ | null
	  data: IRecipeData | null
	  }
	}
	
	This will require checking for a timeout in the UserService.

(public) DELETE /	Used to delete a recipe or a recipe version.
	
	Request {
	  Body {
	  id: string
	  version: string | null
	  }
	}
	
	This can require checking authorization in the AuthService.
	
	This can require updating a collection in the UserService.

(public) GET /metadata	Used to get a recipe's metadata.
	
	Request {
	  Body {
	  id: string
	  }
	}
	
	Response {
	  Body {
	  metadata: IRecipeMetadata
	  }
	}

Interfaces

IRecipeMetadata	{
	  owner: string
	  visibility: 'public' | 'private' | 'unlisted'
	  versions: string[]
	  latest: string
	}

IRecipe	{
	  owner: string
	  visibility: 'public' | 'private' | 'unlisted' | 'deleted'
	  references: number
	  rating: number
	  data: IRecipeData
	}

IRecipeData	{
	  title: string
	  text: string
	  // to-do
	}

API Dependencies

UserService	When a new recipe is created, the UserService must be informed so it can update is owned recipes collection. This only happens on the first creation, and on final deletion.
AuthService	Most checks should be completable by checking the user id in the JWT, but if a user tries to access to complete an action not generally allowed (with nonmatching owner id), then the AuthService will be required for getting a user's role data before deciding whether the user should be permitted to do the action.

UserService

The UserService will handle all user related data. Primarily, this is just a collection cookbooks and recipes. Profile page information will also be included here, as well as a list of the users one is following.

When a user is created (which happens at registration), a default cookbook will be created called saved which is the default location for all newly saved recipes.

If a user is deleted, all their owned cookbooks will also be deleted, and their owned recipes will be marked for deletion.

API Endpoints

(public) GET /	Used to get information about a user.
	
	Request {
	  Body {
	  id: string
	  }
	}
	
	Response {
	  Body {
	  user: IUser
	  }
	}

(public) UPDATE /	Used to update a user's page data or timeout. Role data is required if setting a user's timeout.
	
	Request {
	  Body {
	  id: string
	  timeout_until: number | null
	  data: IUserPageData | null
	  }
	}
	
	This can require checking authorization in the AuthService.

(public) DELETE /	Used to delete a user completely, requires reauthentication.
	
	Request {
	  Body {
	  id: string
	  username: string
	  password: string
	  }
	}
	
	This will require deleting recipes in the RecipeService.
	
	This will require deleting cookbooks in the CookbookService.
	
	This can require checking authorization in the AuthService and deleting authentication entry.

(public) GET /following	Used to get the list of users you are following.
	
	Request {
	  Body {
	  id: string
	  }
	}
	
	Response {
	  Body {
	  following: string[]
	  }
	}

(public) PATCH /following	Used to add or remove a user from the following list.
	
	Request {
	  Body {
	  id: string
	  add: string | string[] | null
	  remove: string | string[] | null
	  }
	}

(private) PATCH /cookbooks	Used to add or remove a cookbook from the cookbooks list.
	
	Request {
	  Body {
	  id: string
	  add: string | string[] | null
	  remove: string | string[] | null
	  }
	}

(private) PATCH /recipes	Used to add or remove a recipe from the recipes list.
	
	Request {
	  Body {
	  id: string
	  add: string | string[] | null
	  remove: string | string[] | null
	  }
	}

Interfaces

IUser	{
	timeout_until: number
	recipes: string[]
	cookbooks: string[]
	following: string[]
	followers: number
	data: IUserPageData
	}

IUserPageData	{
	description: string
	// to-do
	}

API Dependencies

RecipeService	Deletion of a user requires deleting the user's recipes
CookbookService	Deletion of a user requires deleting the user's cookbooks
AuthService	Most checks should be completable by checking the user id in the JWT, but if a user tries to access to complete an action not generally allowed (with nonmatching owner id), then the AuthService will be required for getting a user's role data before deciding whether the user should be permitted to do the action. Deleting the user requires deleting authentication data.