Natalia Palej A00279259

Software Design with Artificial Intelligence for Cloud Computing

13/03/2025

SOA4 Project

A dog sitting in the grass

AI-generated content may be incorrect.

Contents

[Introduction 2](#_Toc193203992)

[Connecting Two Services 2](#_Toc193203993)

[CRUD Operations 2](#_Toc193203994)

[Post: Add New User 2](#_Toc193203995)

[Put: Edit Existing User 4](#_Toc193203996)

[Delete: Delete User 5](#_Toc193203997)

[Caching with ETags 6](#_Toc193203998)

[Conclusion 8](#_Toc193203999)

# Introduction

For my project I have created two services that serve as Dog Adoption Management App. It is a microservice-based application built with Java and Spring Boot. The application consists of two separate services where each service runs independently on separate ports:

|  |  |  |
| --- | --- | --- |
| **Service** | **Description** | **Port** |
| **dog-service** | handles all dog related data (CRUD for dogs) | 8081 |
| **user-service** | handles all user related data (CRUD for users) and allows users to adopt dogs | 8080 |

Both services communicate using Feign Client to enable dog adoption functionalities. Users table in user-service is called “person” because “user” turned out to be a reserved keyword in the H2 database. IDs for both users and dogs are auto-generated.

# Connecting Two Services

To allow communication between user-service and dog-service, FeignClient was uesd. This allows user-service to make REST calls to dog-service internally. This client enables user-service to fetch dog details from dog-service.

A close-up of a computer code

AI-generated content may be incorrect.

Figure DogClient.java

**@FeignClient** - used to declare REST client.

**name** - refers to dog-service, which is the producer

**url** - specifies the address and port where dog-service is running

# CRUD Operations

Both services have CRUD functionality implemented for their respective entities – Users and Dogs. Since both services share similar CRUD logic, I will focus on demonstrating User entity as an example.

## Post: Add New User

The code for adding user is handled by Post endpoint in UserResource.java.

**@PostMapping** annotation maps HTTP POST requests. User object is then saved via *userRepository.save(user)*. Method returns 201 Created status with created user.

A computer code on a white background

AI-generated content may be incorrect.

New users can be added using the form on the right side of the Users table. On the example below I have filled in the details with “Test” and no dog ID (as it can be null for users, since they may not have any dogs assigned to them yet).

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.After clicking “Save User” button, we can see that the user was created and status 201 was returned.

A screenshot of a computer screen

AI-generated content may be incorrect.Then, after refreshing Users table, we can see new user was added to it and holds ID 7. We can also see that new ETag was created as Users table was modified and no longer matched cached version.

## Put: Edit Existing User

The edit functionality is implemented using **@PutMapping** which updates an existing user. To ensure that specific user is updated, we pass user’s **“/{id}”** to the endpoint and in the method.

First, we verify if the user exists. If it doesn’t, 404 Not Found would be returned in response. Since users can adopt dogs, before updating the user, we need to check if dog ID that the user is trying to adopt exists in the dog-service. This is done by calling dog-service using *dogClient.getDogById(dogId)*. If the dog ID doesn’t exist, 400 Bad Request is returned to prevent assigning non-existent dogs.

A screenshot of a computer code

AI-generated content may be incorrect.

When “Edit” button is clicked, the form appears on the right hand side of the table. It is also pre-populated with that specific user details. In the example below, I am updating user ID 1 and I am updating it with Dog ID - from N/A to 1.

A screenshot of a computer

AI-generated content may be incorrect.

If both checks pass, the user is updated in the database using *userRepository.save(updateUser)* and 200 OK response status is returned. ETag value is automatically updated when the table changes, so after the user is updated, new ETag is generated and also shown in the response.

A screenshot of a computer

AI-generated content may be incorrect.

## Delete: Delete User

Delete user is implemented with **@DeleteMapping** method. Before deleting user, there is check whether the user exists using *findById()*. If not found, it returns 404 Not Found.

Then, we need to handle the fact that user may have dog assigned to themselves. If the user is deleted, the service makes a call to dog-service to mark the dog as available again by *updateDogAvailability(user.getDogId(), true)*. A screenshot of a computer code

AI-generated content may be incorrect.

A black and white screen with white text

AI-generated content may be incorrect.In the example below, we will try to delete the “Test” user that we have created earlier. Before deleting the dog, user gets an alert to confirm it or cancel.

A screenshot of a phone

AI-generated content may be incorrect.Once confirmed, *userRepository.deleteById()* is called to delete the user and 204 No Content response is returned.

A screenshot of a computer screen

AI-generated content may be incorrect.Upon refreshing Users table, the user in question is no longer in the table and once again, new ETag was created.

# Caching with ETags

To optimize performance and avoid unnecessary data transfer, caching was implemented using ETags with ShallowEtagHeaderFilter. Its functionality was partially showed above when users table was being fetched and new ETag was created after the table was modified.

The ETag mechanism works by generating unique identifier (ETag) for specific response. When the same request is made again, client sends ETag in the If-None-Match header. If data has not changed, server responds with 304 Not Modified and returns cached version of that request.

To enable caching, new class was created within user-service called CacheConfig. This automatically generates ETags for all “*/users*” responses.

A close-up of a computer screen

AI-generated content may be incorrect.

Figure CacheConfig.java

**@Configuration** annotation marks this class as a configuration class where beans are defined. It tells Spring to process this class for bean definitions at runtime.

**@Bean** annotation indicates that the *shallowEtagFilter()* method returns a Spring-managed bean. This bean will be registered in the application context.

**FilterRegistrationBean** is used to register a servlet filter (ShallowEtagHeaderFilter) and apply it to specific URL patterns.

**ShallowEtagHeaderFilter** is a built-in Spring filter that creates and validates ETags for web responses. It generates an ETag hash based on the content of the response.

**frb.addUrlPatterns("/users")** specifies that this filter should only apply to the “**/users**” endpoint. Every time data from */users* is fetched, the ETag will be checked to determine if reloading the full data is necessary.

If the same request is made and the data has not changed, the server responds with HTTP 304 Not Modified, meaning no new data is sent and cached version is used. If the data has changed, new ETag is generated and the updated response is sent.

Changes were also needed in JavaScript. In *fetchUsers()*, the ETag is stored and included in subsequent requests. If the data hasn’t changed, 304 Not Modified response is returned and cached data is used.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure scripts.js

Here is example where cached response is returned, showing that ETags work as expected. The table uses cached data as no updates were detected.

A screenshot of a computer

AI-generated content may be incorrect.

# Conclusion

The Dog Adoption Management App successfully demonstrates working microservices architecture using Java and Spring Boot. It supports full CRUD operations, cross-service communication and efficient caching with ETags to optimize data fetching. This project reflects real-world microservice principles, including service independence, RESTful communication and performance optimization through caching.