CLIENT-SIDE TECHNOLOGIES

Chaitanya Arora - BWTFX8

GAME OF THRONES

Date:23rd May 2023

Supervisor: Dr Mohammad Saleem

Github Repo: https://github.com/carora2209/GOTE.git

Introduction of the application and description of the main functionality:

The goal of the project is to display information about the famous book series Game of Thrones, it allows users with different functionalities starting from checking the various books, characters and houses. The author of the series – George R.R. Martin – has created a whole universe with over 4000 characters. This project can be helpful for anyone who is an enthusiast of this space and can help him navigate and explore.

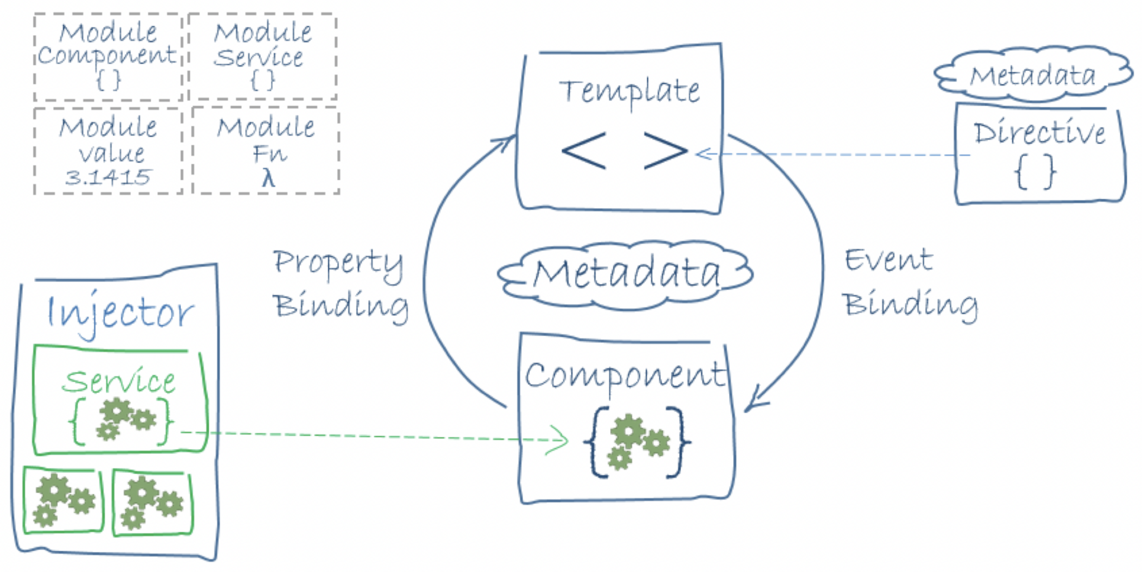
Overall Workflow and Features of the Project:

1. The project starts with the main page consisting of headers and a list of all books presented as a list providing options to view more details.
2. After this, the user is capable of traversing around to check specific things.
3. The user is provided with the options in a header which is a reusable element used on all pages, it helps in easier navigation to different elements thereby maximising user experience.
4. Specific buttons in the header lead to different pages which handle a specific functionality.
5. Every element of the list is clickable and allows one to view the full details of the entity.
6. Overall the application is implemented in an interactive and ergonomic way.

Description of Architecture and the Introduction to main components:

The project makes use of Angular for the project development, Typescript as the programming language because of its versatile nature which makes it extremely easier for us to manage the application. Moreover with its growing popularity in the JS community, it seemed like the perfect candidate.

In Angular projects, it is typically more convenient to adhere to a standardized project architecture. This architecture follows a modular structure and emphasizes the separation of logic and interfaces. By organizing the project in this way, development and maintenance can be done more effectively. The diagram below depicts the architecture of this particular project.



Component classes represent the different sections which are used on different pages, each of them displaying a separate aspect. Each component has a set of files, those are namely the HTML, CSS, and .TS files, handling the content, styles and structure of the component.

Main Components are

- Book item- this lies on the first page which lists different book names and gives the option to view more details
- Houses: this represents the list of houses that we see when we click on the "houses" Button on the header
- House-Details: this represents the particular details of a house namely region, world, titles etc.
- Header - this is a common header which is present on every page
- Character item- this presents the list of characters who were there in that specific movie
- Details- this page refers to the overall page that we reach when we call the "view details" of a book
- Display-Book: this project presents the details of the book namely the name, author, ISBN number etc
- Scroll- this refers to the functionality that is offered to scroll the list of characters so that a user can browse if they are a big number
- Display Character- this displays the details of a particular character

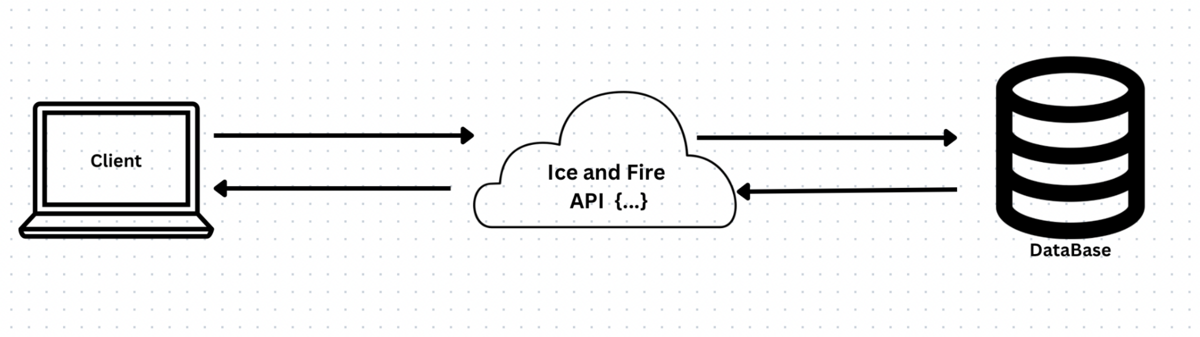
- Character-Details: this refers to the overall all page which we call the display character component
- Books: This component makes the call to the book item page which lists all the books
- Display-all-character: this calls the display character functionality
- Display-all-houses: these calls display houses component

A list of classes created by you with a brief description of their tasks:

Every component has a class with the same name of it (e.g. CharactersComponent class) that implement OnInit. Their functionality was described above. Plus for every element there is a logic-proving class – called service, they are: CharacterService class, BookService class and HouseService class.

Description of Client-Server Communication:

In order to reach the remote server we are making use of the HTTPClient which has to be imported and is initialised in the constructor of every component. After which it is leveraged to perform a GET operation on the specific URL that the component deals with. An observer design pattern is used to automatically update information on every change (part of RxJS). The response is an untyped JSON object, so special types for character, book and house were created to specify it. To get specific elements by id the Subscribe()method was used that connects the observer to observable events.



Select and describe the whole process of how a specific API call is sent to the server and how the results are received and displayed! Include source code if necessary

The example described below will explain the API call of the book service.

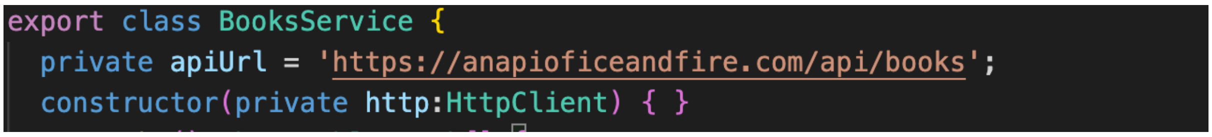
Step1:

```
import { Injectable } from '@angular/core';

import {map, Observable } from 'rxjs' ;

import {HttpClient} from '@angular/common/http';
import {Book} from '../Templates/Book';
```

The first step involves importing the necessary modules which will be required in the code further:

1. The HTTP client is a building angular module used to make HTTP calls to the server.
2. An observer design pattern is used to automatically update information on every change (part of RxJS).
3. The 'Injectable' decorator is used to mark a class as a dependency injection token so that it can be injected into other classes.
4. Importing the template of the book

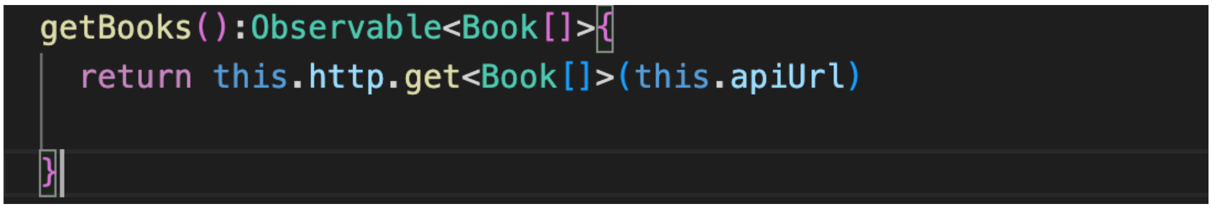
Step2:



```
export class BooksService {
  private apiUrl = 'https://anapioficeandfire.com/api/books';
  constructor(private http:HttpClient) { }
```

We store the correct API URL which will be used in the HTTP request that we make, and we initialise the HTTP client in the constructor

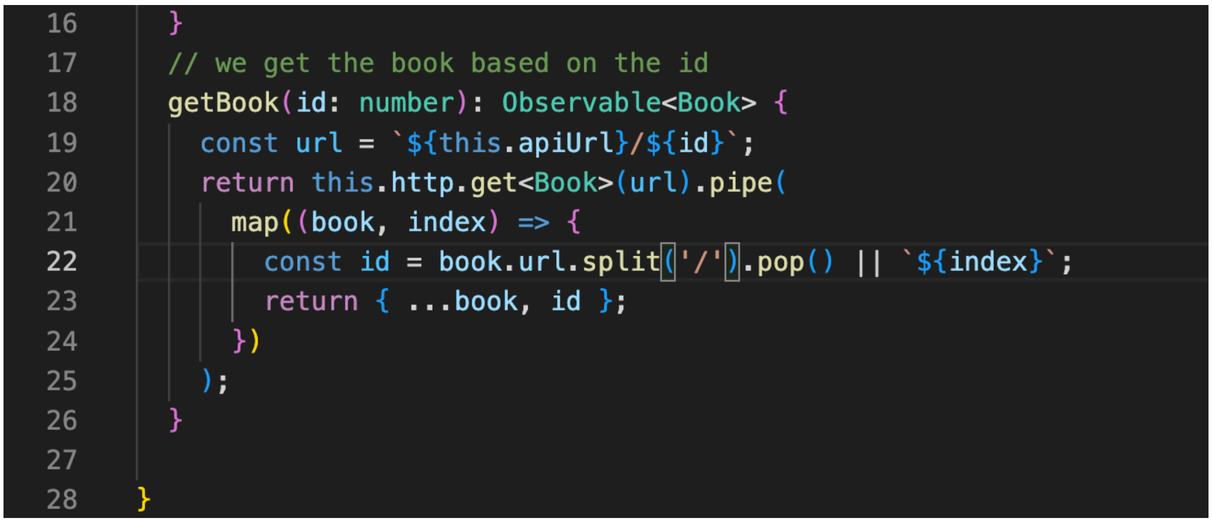
Step 3:



```
  getBooks():Observable<Book[]>{
    return this.http.get<Book[]>(this.apiUrl)

  }
```

This is when we are using the observer design pattern to automatically update on every change

Step4:



```
  }
  // we get the book based on the id
  getBook(id: number): Observable<Book> {
    const url = `${this.apiUrl}/${id}`;
    return this.http.get<Book>(url).pipe(
      map((book, index) => {
        const id = book.url.split('/').pop() || `${index}`;
        return { ...book, id };
      })
    );
  }

}
```

This method makes an HTTP GET request to retrieve a book object from a specified URL. It then uses the map operator to transform the emitted value by extracting the id from the book object's URL and returning a new object with the id property added. The result is an Observable of type Book that emits the modified book object.