Connecting to a REST Endpoint

Choosing a Backend

- The default data format is JSON, which can be used with pretty much any backend.
  - Java Spring uses JSON by default
  - Python Flask can be configured for JSON with the Flask-JSON library
- The api needs to return a text JSON data via HTTP GET for specified endpoints
- Here's an example route setup:

HTTP VERB	URL	Action
GET	/api/rooms/	Get All Rooms
GET	/api/rooms/123	Get the room with id 123
POST	/apt/rooms	Add a Room
PUT	/api/rooms	Update a Room

One important thing before we get into accessing REST api from angular: Never store plaintext passwords in Angular. Since all Angular code is run through the browser it wouldn't be hard to crack.

Connecting Angular

- First thing to do is import `HttpClientModule` from `@angular/commons` in app.module.ts
- Then we can import HttpClient into the constructor and use that to perform HTTP calls

```
constructor(private http: HttpClient) {
  console.log(environment.restUrl);
}

getUser(id: number): Observable<User> {
  return this.http.get<User>(environment.restUrl + '/api/users/' + id);
  }
```

The get method above accepts a generic type as a hint to the compiler. getUser is actually returning a JS Object with fields that match the user class, but it's not actaully an instance of a User.

Configuring CORS

- Of course, it's never as easy as that. If we tried to run the above service method we would get an error something like: `[your-host]/api/users/[id] has been blocked by CORS policy: No 'Access-Control-Allow-Origin header is present'`
  - This means the server was reached but the reply was blocked.
- CORS is a security policy implemented by browsers to protect users against what are known as 'hijacking attempts'
  - Basically JavaScript calls a server to get data to display on the page. As long as it comes from the same server as the JS, we're OK. But if the call is to a different server (called a **cross-origin request**) the request is blocked.
  - To allow requests to different servers we'll need to enable **Cross-Origin Resource Sharing** to give explict permission
  - Also the port number is enough to define a unique server. Eg. localhost:4200 and localhost:8080 are concidered different servers
  - CORS is enabled and configured on the Backend
- The following example is using Java Spring, but this should exemplify the fundemental idea for any backend.

```
@Configuration
public class CORSConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowedOrigins("http://localhost:4200");
    }
}
```

Pre-Processing REST Data

You could put all the preprocessing code in the dataService class subscription within the ngInit method, but that can lead to sloppy code. Instead use the pipe method within the dataService class function which is called for the same effect.

In the below example we create a JSON type user from the User object recieved from the api

```
export class User {
  id: number;
  name: string;

  static fromHttp(user: User): User {
    const newUser = new User();
    newUser.id = user.id;
    newUser.name = user.name;
    return newUser;
  }
}
```

```
// some.component.ts
  ngOninit {
  this.dataService.getUser(13).subscribe(
      next => {
        console.log(next);
        console.log(typeof next);
        console.log(next.getRole())
    });
  }
```

```
//data.service.ts
  getUser(id: number): Observable<User> {
    return this.http.get<User>(environment.restUrl + '/api/users/' + id)
      .pipe(
        map( data => {
          return User.fromHttp(data);
        })
      );
```

Dealing with Slow and Unresponsive Connections

Sometimes the data will not be available right away. In this case it would be nice to display something like "Loading, please wait."

To do this we create a boolean in the typescript file for the component loading the data

```
  message = "Loading... Please wait."
  loadingData = true;

  ngOnInit(): void {
    this.dataService.getRooms().subscribe(next => {
      this.rooms = next;
      this.loadingData = false;
    });
  }
```

And then we can query that parameter from the HTML

```
<div *ngIf="loadingData">{{message}}</div>
```

But what if the entire backend is down?

We can add an additional parameter to the data service to catch errors.

```
  message = "Please wait... getting the list of rooms"

  ngOnInit(): void {
    this.dataService.getRooms().subscribe(next => {
      this.rooms = next;
      this.loadingData = false;
    },
      (error) => {
        console.log('error', error);
        if (error.status === 402) {
          this.message = "Sorry - payment is required to use this application."
        } else {
          this.message = "Sorry, something went wrong. Please try again later.";
        }
      });
```

For helpful error messages, make sure you properly set the return status code from the back end. Then you can customize the response based on the status code. By default a nonresponsive server returns status code 0 to the browser.

Retrying on error

The best way to do this is by creating a counter and recursively calling a load data function.

```
reloadAttempts = 0;


loadData() {
  this.dataService.getRooms().subscribe(next => {
      this.rooms = next;
      this.loadingData = false;
    },
    (error) => {
      console.log('error', error);
      if (error.status === 402) {
        this.message = 'Sorry - payment is required to use this application.'
      } else {
        this.reloadAttempts++;
        if (this.reloadAttempts <= 10) {
          this.message = 'Sorry, something went wrong. Trying again...';
          this.loadData()
        } else {
          this.message = 'Sorry, something went wrong. Please contact support.'
        }
      }
    });
}
```

A Common Bug

Say you save some data and it redirects to the view page. The data might not be immedatly available if there is a delay on the backend, so the console will get an error like 'Object is null'.

To prevent this we can use the null check op in TypeScript:

```
<table>
  <tr>
    <td>id</td><td>{{ room?.id }}</td>
  </tr>
```

```
    <tr>
      <td>name</td><td>{{ room?.name }}</td>
    </tr>
    <tr>
      <td>location</td><td>{{ room?.location }}</td>
    </tr>
  </table>
  <h4>This room can accomodate:</h4>
  <table *ngIf="room.capacities">
    <tr *ngFor="let layoutCapacity of room.capacities"> <!-- repeat the tr for each capacity -->
      <td>{{ layoutCapacity.layout }}</td><td>{{ layoutCapacity.capacity }}</td>
    </tr>
  </table>
```

So it will only display if the variable is not null. Note that this is one of the few situations to use this, it's generally not good practice.

Confirming Action

To confirm a delete, update, etc. It is very simple to accomplish in Angular

```
  deleteBooking(id: number): void {
    const result = confirm('Are you sure you wish to delete this booking?');
    if (result) {
      this.message = 'Deleting data...';
      this.dataService.deleteBooking(id).subscribe(
□ ...
```

Pre-Fetching Data and Using Resolvers

Concider the following component code for a calander app:

```
  ngOnInit(): void {
    this.dataService.getRooms().subscribe(
      next => this.rooms = next
    );
    this.dataService.getUsers().subscribe(
      next => this.users = next
    );

    const id = this.route.snapshot.queryParams['id']
    if (id) {
      this.dataService.getBooking(+id).subscribe(next => {
        this.booking = next;
        this.dataLoaded = true;
        this.message = '';
      });
    } else {
      this.booking = new Booking();
      this.dataLoaded = true;
      this.message = '';
    }

  }
```

We have 3 different data services making calls asyncrounsly. This could lead to problems when the page is loading, as the users/rooms might be unavailable when the booking is loaded. To solve this we could nest each data service call within the `next` block or increment a counter, but this makes the code syncrounous and slows down the user experience.

The solution is to create a resolver. A resolver is an object which resolves an observable. The resolver does the subscribing and then waits for the data to become available. In the above case we extract the subscription component for users and rooms from the calander component and add them to two new services: prefetch-rooms.service.ts:

```
import { Injectable } from '@angular/core';
import {Observable} from "rxjs";
import {Room} from "./model/Room";
import {Resolve} from "@angular/router";
import {DataService} from "./data.service";

@Injectable({
  providedIn: 'root'
})
export class PrefetchRoomsService implements Resolve<Observable<Array<Room>>>{

  constructor(private dataService: DataService) { }


  resolve() {
    return this.dataService.getRooms();
  }
}
```

prefetch-users.service.ts

```
...
Resolve<Observable<Array<User>>>{


  constructor(private dataService: DataService) { }


  resolve() {
    return this.dataService.getUsers();
  }
}
```

Then in app.module.ts we add the following to the routes:

```
const routes: Routes = [
  { path : 'editBooking', component : CalendarEditComponent, resolve : {rooms : PrefetchRoomsService, users :
PrefetchUsersService}},
  { path : 'addBooking', component : CalendarEditComponent, resolve : {rooms : PrefetchRoomsService, users :
PrefetchUsersService}},
  ...
```

So now when we navigate to addBooking or editBooking the resolvers are going to be set up so our data is available within the route, although it is hidden. Now we can extract the data into the edit-calander.component.ts as follows:

```


  constructor(private dataService: DataService,
              private route: ActivatedRoute,
              private router: Router) { }


  ngOnInit(): void {
    this.rooms = this.route.snapshot.data['rooms']
    this.users = this.route.snapshot.data['users']
    const id = this.route.snapshot.queryParams['id']
    if (id) {
      this.dataService.getBooking(+id)
        .pipe(
          map (booking => {
            booking.room = this.rooms.find(room => room.id === booking.room.id);
            booking.user = this.users.find(user => user.id === booking.user.id);
            return booking;
          })
        )
        .subscribe(next => {
        this.booking = next;
        this.dataLoaded = true;
        this.message = '';
      });
    } else {
      this.booking = new Booking();
      this.dataLoaded = true;
      this.message = '';
    }

  }
```

Revision #2
Created 16 April 2022 23:19:27 by Elkip
Updated 16 April 2022 23:40:55 by Elkip