Making Http Requests

How Does Angular Interact with the Back-End?

- Angular doesn't directly enter the database
- Instead, we send **HTTP requests** to and from a server
  - In this case, the server is a REST API
  - We don't get new HTML back from **URL's**, but **data**

The Anatomy of an Http Request

- **HTTP Verb** → POST, GET, PUT, DELETE, etc
- **URL (API Endpoint)** → /posts/1
- **Headers** → { "Content-Type": "application/json" }
- **Body**
  - Attached to **POST**, **PUT**, and **PATCH**
  - { title: "New Post" }

Sending a POST Request

- We need **HttpClientModule** in **AppModule**'s **imports** array
- We must inject **HttpClientModule** into the component to use it
- **POST**
  - Used to send data → **BODY**
  - Parameters
    - **URL** → The **API endpoint**
    - **BODY** → The **content** we're sending over
- HTTP requests are managed by **Observables**
- What happens if we don't subscribe to a request?
  - Angular assumes that nobody wants the response
  - The request is never sent
- **POST** requests are always sent in pairs
  - The **first** is of type **OPTIONS** → Checks if the post is permitted for sending
  - The **second** is the actual **POST** request

GETting Data

- **Fetches** data from the back-end
- Only takes **one argument**
  - **URL** - The **API endpoint**
  - Since we're not passing data to the back-end, there is no body

Using RxJS Operators to Transform Response Data

- **Pipe** → Let's us funnel observable data through multiple **RxJS operators**
- **Map** → Applies a given function to each value **emitted** by the source **Observable**, and **emits** the resulting values as an **Observable**

Using Types with the HttpClient

- Angular doesn't know the **innate type** of **responses**
- We should explicitly define types when handling data
- To simplify types, we should create and use **models**
- **GET** and **POST** are generic methods → We can use **<>** to define the response type

Showing a Loading Indicator

- Create a **loading field** in the component, setting it to **false**
- Set it to **true** when fetching posts, then back to **false** when you **subscribe** to whatever you have **fetched**
- Use **ngIf** to show the loading information in the template

Using a Service for Http Requests

- It's good to **outsource logic/functionality** to **services**, leaving component code to template-related material
- Create a **new service**, ensuring to **inject HttpClient**
- Refer to the **service** and its data/API functions by **injecting** it into the **host component**

Services and Components Working Together

- We should **define** and **invoke** our **APIs** as **functions** within the **service**
- In our **component**, we should **call** and **subscribe** to the **service functions**

- The functional **data** that pertains to the **component's template** should be in the **component**, not the service
- It's okay to subscribe inside of the service if the component doesn't care about whether the request completes

Sending a DELETE Request

- Like **GET**, has no **BODY** parameter
- Must return the observable to inform the component

Handling Errors

- **Subscribe**'s second parameter handles errors
- This is a **callback function**, and it should perform the necessary instructions for preserving your program
- It's good to **alert users of errors**, as well
- **Errors** with **API**'s tend to return a lot of **data** → We can view this in the → **Inspector Menu**

Using Subjects for Error Handling

- When **subscribing** in the service, you should handle errors with **Subjects**
- This **Subject** is stored within the **service**
- We **subscribe** to the **subject** in the **component** to interact with or present errors
- Again, we should **unsubscribe** from the error when we finish using the component

Using the catchError Operator

- **catchError** is an **RxJS operator** that automatically calls code upon meeting an error
- This is meant for **generic error-handling**
- To throw the error, we return a **throwError Observable**

Setting Headers

- Must import **HttpHeaders**
- We sometimes need headers for things like **authorization**, **content-type**, etc

- We set **headers** by inserting them as the **last argument** for each response type
  - **POST** → **Third** argument
  - **GET** - **Second** argument
- Headers are represented as **JavaScript objects** with **key/value pairs**

Adding Query Params

- Must import **HttpParams**
- You can add **HttpParams** by inserting it as a **key/value pair** within the final argument
- **Setting** params is done using the **set** method
- One can also create an instance of **HttpParams** and feed it **multiple params** using the **append** method
  - Must override the existing variable for all appends

Observing Different Types of Responses

- It's possible to observe the **entire HTTP response** instead of merely the unpacked body data
- We do so by passing the **observe key/value pair** into the **final argument's JavaScript object**
- The default value is **'body'** → What we always get
  - We can also use **'response'** to capture the entire response, including **headers**, **status**, **type**, etc
  - Another is **'events'**
    - Requires the **RxJS tap operator** → Let's us execute code without altering the response
- Use **dot notation** to access **response attributes**

Changing the Response Body Type

- We can define the **responseType** in the **final argument's JavaScript object**
- This includes types such as **json** (default), **text**, etc

Introducing Interceptors

- We typically assign **different headers** to each request because not all requests are necessarily the same
- **Interceptors** allow us to attach **specific headers** to **all HTTP requests**
- Example → Adds auth-related headers for all authentication requests
- You can create **interceptors** as **service** files
- Intercepts implement the **HttpInterceptor** interface
  - Forces us to employ the **intercept()** function
    - Arguments
      - First - **Request Object** → Type of **HttpRequest<>**
      - Second - **Next** → Type of **HttpHandler**; Function that forwards the request through the rest of its journey
- We must provide the service to the **AppModule providers** array
  - Inserted as a **JavaScript object** → **{ provide: HTTP_INTERCEPTORS, useClass: <interceptor>, multi: true }**

Manipulating Request Objects

- Within our **intercept** function, we can clone and modify a request using **<req>.clone(<modifications>)**
- We can change **headers**, **URLs**, etc within **<modification>**
- This **cloned request** should be stored into a **variable**, which is then passed into the returning **next.handle()**

Response Interceptors

- **next.handle** actually returns an **Observable** which lets us interact with the response
- We do so using the **RxJS pipe operator**

Multiple Interceptors

- It's **possible** to employ **multiple interceptors**
- Be careful with how you **order interceptors** in **AppModule** as that's the order in which they're executed
- Similar to before, add a new object to **AppModule's providers** array