HttpClient

communicating with a server

Communicating with server

- We want to connect our web application to a database
- we can’t connect directly from the frontend app to the database
- we usually will have a backend server that will be connected to a database and we will grab the data from the database via the backend server
- we will usually communicate with our server via http protocol
- It’s common to interact with the backend via REST
- we want to send the request to the server via AJAX
- We will get the response async
- if the browser there 2 ways to send AJAX request:
    - XMLHttpRequest
    - fetch
- When using angular we have a service that wraps XMLHttpRequest and allows us to send ajax request and interaction with the request response will be with observables

@angular/common/http

- the angular service that wraps XMLHttpRequest is called **HttpClient**
- the module of **HttpClient** is called **HttpClientModule**
- to use the **HttpClient** we have to add **HttpClientModule** to the **imports** array of the module metadata
- Let’s install **HttpClientModule...**

Our first request

- let’s try and make a request to grab all the tasks from our todo rest server
- we will send a get request to the url: **https://nztodo.herokuapp.com/api/task/?format=json**
- we will get all the tasks and print the title
- the first argument is the url of the request
- the second argument is optional options that you can send to customize the request and the response you are getting
- Let’s go over the other methods that are commonly used with **HttpClient**
  - **put**
  - **post**
  - **delete**

calling request with generic type

- from the service declaration we can see that there are different methods call that will request the data differently
- most of the methods vary in the way the response will return
- some methods will allow you to subscribe to request events
- the default methods will return a json response
- you can also add a generic type to the request to enforce the structure of the json response
- supplying an interface as the json structure will help us reduce bugs that happen from json returning not in the format we expect

Handling error response

- two types of error can occur during sending a request
  - an error that cause the request to not even reach the backend
  - server returns an error
- you deal with those error by subscribing an error callback in the subscribe
- the error callback will be called with **HttpErrorResponse**
- on the first type of error we will have an **error** attribute of type exception
- on the second the error attribute will contain the string of the response from the server
- let’s try to deal with the two types of error

Grabbing the response

- when calling the methods in **HttpClient** , by default you get the json object the server returned
- sometime you will want to also get the full response
- you can pass as the second argument (or third if the method should contain body) **{observe: ‘response’}**
- the response will have a **body** attribute of the like the generic type you send
- you can observe also the body or events

request with body

- when creating a new task with post or updating a task with put, you will need to send a body with the request
- the body will be the second argument of the method and will be a dictionary with the key values you want to send
- let’s send a post request to create a new task

Authentication & Authorization

- to make our application secure, we have to decide on every data resource, who can read it.
- so when the user requests data we need to ask two questions:
  - who is the user (authentication)
  - is the user allowed to access the resource he is requesting (authorization)
- to help us answer those questions we have to assign a token to a user
- when the user logs in our application the server will return the token (or create a new one and return the new one)
- on every request the user will send the token with the request
- it’s common to send the token in **Authorization** header
- let’s try and send the authorization token with our request

XSRF

- Stands for Cross Site Request Forgery
- tricks a user to make an action on a site that he does not intend to do
- example of abuse:
  - a hacker studies a vulnerable site and examine the requests the site is making
  - the hacker sending a mail with a link to a malicious site
  - the malicious site on your behalf sends a request to the backend server
  - since you are logged in the site the request will be accepted
- One way to avoid it is the server created XSRF-TOKEN on the initial load
- that token is sent with every request
- angular makes the XSRF easy for us by just loading the **HttpClientXsrfModule**
- the module will only work for relative requests
- headers will not be sent on get
-

Interceptors

- we saw an example of sending Authorization header
- for best practices it’s better not to repeat the headers action on every request but add authentication interceptors
- interceptors sit between the application and the backend
- interceptors can transform the request before sending it to the server
- interceptors can transform the response before your application can see it
- an Interceptor is a class that implements: **HttpInterceptor**
- **HttpRequest/HttpResponse** are immutable
- let’s create an auth module that adds authorization and XSRF
- more things we can do with interceptors:
  - log exceptions
  - time requests
  - caching requests

Summary

- communication with backend is a major part of a web application
- angular makes it easy for us with **ClientHttpModule**
- we use the **ClientHttp** to make requests
- we use interceptors to add repetitive tasks with the requests and responses
- we remember that a secure web application does the following
    - limit resources in the backend based on authorizing users
    - attaching token authentication to requests
    - protecting against CSRF
    - the requests should be sent over SSL