HttpClient

Server communication & security

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)
- How can we authenticate/authorize the user performing the request?

JWT

- stands for **Json Web Tokens**
- Json based open standart for creating access tokens
- The token is usually signed by private key in the server and only the server can decrypt it
- with JWT we can authenticate and authorize a user
- JWT contains claims about the token like privileges or date issued

JWT Structure

- JWT generally have 3 parts
  - header
  - payload
  - signature
- The header identifies which algorithm to use
  - **header = ‘{“alg”: “HS256”, “type”: “JWT”}’**
- The payload contains the claims to make, we can use this data to know the user that is making the request
  - **payload = ‘{“loggedInAs”: “Admin”, “iat”: 1422777777}’**
- The signature is calculated as the following:
  - **key = ‘secretkey’ // secret string of your choosing**
  - **unsignedtoken = encodeBase64Url(header) + ‘.’ + encodeBase64Url(payload);**
  - **signature = HS256(key, unsignedtoken);**
- So the token is the three parts seperated with ‘.’:
  - **encodeBase64Url(header) + ‘.’ + encodeBase64Url(payload) + ‘.’ + encodeBase64Url(signature)**

Login page

- Practicing authentication and authorization and to show how to do things properly we will implement a simple login page
- we will implement the frontend and backend with express js
- The frontend part containing a login component with a form to input email and password
- the email and password will be send via post request to express server we will build that will authenticate the credentials and if success will create and send a JWT token
- we will limit read resources only to authenticated users

Express serve angular

- let’s start by creating a simple server that will serve our angular app
- this time we won’t do SSR and only serve the static files

Express task api

- our server won’t be connected to a database
- we will hardcode data
- we will create an api in the url **/api/task**
- this api will return an array of hardcoded tasks

Express create JWT

- we will create the JWT using the package: **jsonwebtoken**
  - **npm install jsonwebtoken --save**
- the method to create the token:

```
const jsonwebtokens = require('jsonwebtoken');
jsonwebtokens.sign({
 id: 1, // we can put the entire user here or just the pk
 email: 'yariv@nerdeez.com'
}, secret, {expiresIn: 60 * 60})
```

- We will create an api for the url: /api/login
- the api will accept post request with email and password in body
- we will also need to add a middleware for body-parser to be able to grab the data from the request
- we will also create a login component in angular to send email and password to the api we created

Restricting the task api

- we want only authenticated users to be able to read our todo tasks
- there is a package called **express-jwt** which grabs the JWT token from the headers
- the package can restrict resources and send 401 on invalid JWT or no JWT
- the package read the JWT token from the header **Authorization**
  - **Authorization: Bearer <token>**
- the package will fill **req.user** with the payload in the token

```
const jwt = require('express-jwt');
app.get(
    '/api/task',
     jwt({secret: secret}),
     function(req, res) {...})
```

Passing the JWT to all the services

- we acquired the JWT token in the login component but now there is a new problem
- We want all communication to the server to include this token
- How can we achieve this?

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
- more things we can do with interceptors:
  - log exceptions
  - time requests
  - caching requests

Login Service

- we will create a service that will query the login api in the server
- the service will hold a **BehaviorSubject<boolean>** to mark after the user is logged in
- the service will have a method to query to login api and will change the subject to true when login is success and will save the jwt token in the localStorage

JWT interceptor

- we will create an interceptor the will grab the jwt from the localStorage and create an authorization header
- The header will look like this:
  - **Authorization: Bearer <JWT token>**
- we will use **req.clone** to clone the request and we will update the headers of the request
- we also need to add our service to the providers
  - {**provide**: HTTP_INTERCEPTORS, **useClass**: JwtInterceptor, **multi**: **true**}

Server request as event stream

- We can look at server request as a data stream
- when we use **HttpClient** we can pass an option arguments and we can ask to view the events
  - **HttpClient.post('/api/login', {email: '', password: ''}, {observe: 'events'})**
- the stream of events is unnoticeable when we are doing regular requests but we can use it when we are dealing with file upload
- let's try and use HttpClient with our express server to create a simple file upload
- we want the server to signal the frontend on upload progress

<input type=”file”>

- <input type=”file” /> gives a control to the user to select files from his computer
- adding attribute **multiple** will let the user choose multiple files
- you can subscribe to the control **onchange** event to listen to file selections
- the files list will be in the property **files** of the dom element where each file is contained a File object

FormData

- FormData can get a key value from forms and send it to the server as a **multipart/form-data**
- you can give a value of **string, Blob** or **File** and it will transformed to a string format
- you can then send the value to the server

HttpClient to upload file

- the **HttpClient.post** can also get **FormData** as the second argument
- in the third argument of the upload progress we can mark that we want to get the events and also the progress
  - { **observe**: **'events'**, **reportProgress**: **true** }
- progress event will contain the following data
  - {type: 1, loaded: 540672, total: 74776120}
- from this we can create a progress percentage

Express - upload server side

- as we seen before a lot of the times, dealing with common problems in express means loading the correct middleware
- there is a middleware that helps us deal with file upload called **express-fileupload**
  - **npm install express-fileupload --save**
- after loading the middleware we will have **req.files** filled with an object containing the key as the name of the key we are sending the file, and the value will be a File object
- the file object contains a method **mv** where we can move the file to a directory or we can grab the file from the memory and upload it to S3

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
-

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