Introduction to Asynchronous JavaScript

Day 02

Whenever you hear this term, asynchronous, immediately think about time or specifically time delays. This term involves how we manage multiple events where one event has to wait for another event to complete. It is especially important in networking and communications but essentially anywhere an event or task involves some kind of a delay. Think about opening a file, accessing a database or even about a button click.

Asynchronous programming involves applying strategies that enables your program to start a task that may take some time to complete, but at the same time be able to respond to other events happeining simultaneously. If you did not manage this properly, the risk is that the application or OS itself may become non-responsive and lead to poor user experience.

Some of the more advanced programming languages like Java, C++ and C# have the ability to spurn multiple threads. JavaScript needs an environment to operate such as the browser or the NodeJS environment. JS therefore is single-threaded and has to manage multiple tasks on one single tread or processing stream. The bootcamp is about how JS uses its environment to manage multiple tasks going on at the same time.

Content for Day02

Part 1 – Promise Methods

The Proimise object comes with several methods like **.all()**, .**allSettled()** and .**race().** Refer to the documentation for more information on these. We will demonstrate just a couple in the bootcamp. The following code will be executed inside of a NodeJS environment, although most of these functions will work with regular JavaScript running in a browser.

So, if you have the same code from Day01 Part7, then continue using it, otherwise all you need for this part is a JavaScript file running in a Node environment. I will create a folder, then add an index.js file in that folder. To run the index file via the Node engine, simply navigate to that folder via a terminal window and type the command `node index`. The result of executing this index file will appear in the terminal window.

1. Since a Promise is just another object in JS, it comes with a few static methods like `resolve()`. We can call this method immediately and pass some data to it and it will just work:

```
const pf1 = Promise.resolve("done");
pf1.then(data=>console.log(data));
```

2. We could copy the two lines and create a different Promise:

```
const pf1 = Promise.resolve("done");
pf1.then(data=>console.log(data));
//
const pf2 = Promise.resolve("mee too");
pf2.then(data=>console.log(data));
```

3. There is a method however that allows us to handle multiple promises, its called the `.all()` method. Here is how it is applied in this situation:

```
const pf1 = Promise.resolve("done");
//pf1.then(data=>console.log(data));
//
const pf2 = Promise.resolve("mee too");
//pf2.then(data=>console.log(data));
//
Promise.all([pf1, pf2]);
```

   Notice that the `all()` method takes a single array that contains the individual Promises

4. `promise.all()` returns a single Promise, so we still need a `then()` method in order to see any result of the array of promises:

```
const pf1 = Promise.resolve("done");
//pf1.then(data=>console.log(data));
//
const pf2 = Promise.resolve("mee too");
//pf2.then(data=>console.log(data));
//
Promise.all([pf1, pf2])
.then((data)=>{
    console.log(data);
});
```

5. Lets add a third Promise with a delay of zero seconds for now:

```
const pf1 = Promise.resolve("done");
const pf3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 0, 2024);
});
const pf2 = Promise.resolve("mee too");
```

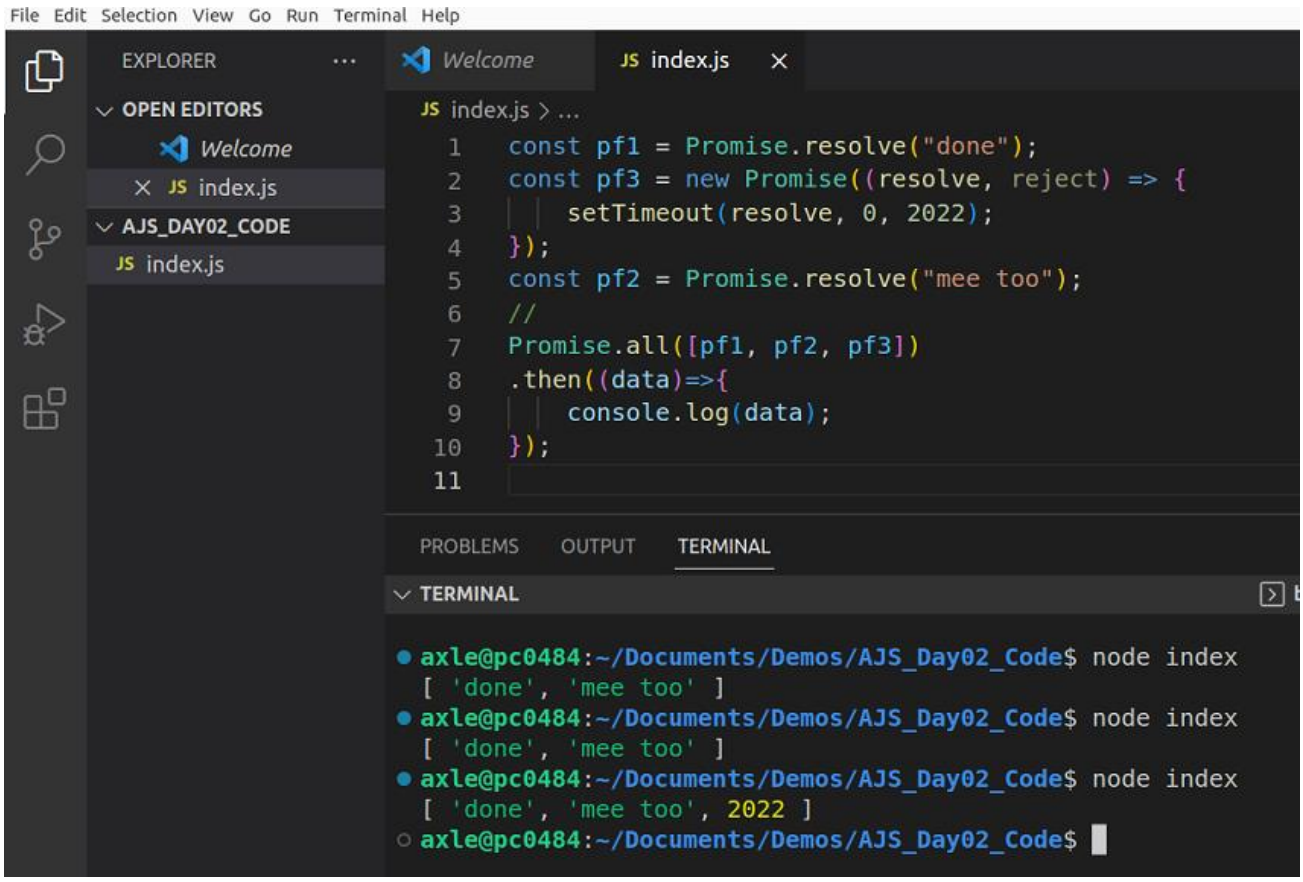
Note: in this case the setTimeout() method is calling the resolve() method after 0 seconds and the value 2022 is being passed into the resolve() method

I removed all the comments for this one, also the data being resolved here is a numeric 2022

6. Now just add this new Promise (pf3) to the array being passed to the all method and everything should work like before just that now we have 3 promisses:

```
const pf2 = Promise.resolve("mee too");
//
Promise.all([pf1, pf2, pf3])
.then((data)=>{
    console.log(data);
});
```

The output should be: ['done', 'mee too', 2024]



7. Now let us add a delay to that third Promise:

```
const pf3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 2000, 2024);
});
const pf2 = Promise.resolve("mee too");
```

In this case we still get the same result but we had to wait 2 seconds

8. Add error handling:

```
Promise.all([pf1, pf2, pf3])
.then((data)=> {
    console.log(data);
})
.catch(err => {
    console.log(err);
});
```

9. Now introduce an error on any of the three Promisses:

```
    setTimeout(resolve, 2000, 2024);
});
const pf2 = Promise.reject("error occured");
//
Promise.all([pf1, pf2, pf3])
.then((data)=>
```

Since the error occurred before any of the promises can be resolved, we only see *error*

10. Now simply change `.all()` to `.any()`

```
 setTimeout(resolve, 2000, 2024);
});
const pf2 = Promise.reject("error occured");
//
Promise.any([pf1, pf2, pf3])
.then((data)=>
```

In this case, if any of the promises resolve before the error occurred, that promise result will be shown. So in my case I see the result of the first promise only.

11. Now simply change `.any()` to `.race()`

```
const pf2 = Promise.reject("error occured");
const pf3 = new Promise((resolve, reject) => {
    setTimeout(reject, 0, 2024);
});
const pf1 = Promise.resolve("done");
//
Promise.race([pf3, pf2, pf1])
.then((data)=>{
    console.log(data);
})
.catch(err=>{
    console.log(err);
});
```

In this case the first Promise that is resolved or rejected is returned and that ends the Promise object, so it works just like any but with subtle differences.

End of section

Part 2 – RxJS Library and Observables

RxJS is an acronym for Reactive Extension for Javascript. It is yet another JavaScript library but one that is centered around reactive programming. At the heart of the library is the Observable. The Observable allows developers to handle asynchronous data calls, callbacks and event-based programs.

The difference between these two concepts, well, promises and observables make requests to a server for data. A Promise will give us all the data at once, assuming the data from the server is ready. An Observable has the ability to emit chunks or packets of data over time. Also, a Promise will ALWAYS return data, no matter what (or error). An Observable, however, will NOT give you any data, unless you *subscribe* to it. This is the famous publish/subscribe pattern.

For this section you do need a package.json file, you may create one yourself or simple run the **`npm init`** command and accept all the defaults. All of this must be done in some folder. You could use the same code from Day02 Part1.

1. In your NodeJS environment, install the library using this command:

```
npm install rxjs
```

2. Make sure you have the *type* setting as *module* in the package.json file:

```
{
  "type": "module",
  "dependencies": {
    "json-server": "^0.17.0",
    "rxjs": "^7.5.5"
  }
}
```

3. Once you have imported *rxjs* into your index.js file, we can create an Observable using the **`create()`** method, one of several methods in this library:

```
import { Observable } from 'rxjs';
const observable = new Observable();
```

Notice that I imported via de-structuring, the Observable package from the library.

4. The constructor of the Observable class takes a function that will engage with a *subscriber* to this Observable so lets prepare for this function:

```
const observable = new Observable( ( ) => {

} );
```

This inner function is basically the *Observer* or listener, it will react to the Observable once the observer is configured.

5. The function that an Observable object takes, is called a subscriber function or an observer. It has three methods that will activate once the Observable object realizes that it has a related observer object listening for data via those methods:

```
const observable = new Observable( ( s ) => {
    s.next("Hello");
});
```

In this case, the subscriber function uses the **`next()`** method to emit a string value

6. As a matter of fact, the subscriber function can emit as much data as it wants:

```
const observable = new Observable((s)=>{
    s.next("Hello");
    s.next("from");
    s.next("Skillsoft");
});
```

At this point, the Observer does nothing, it has no subscriber as yet. The only thing we did was to configure it.

7. Now on to the subscription. The Observable will NOT activate until it sees a *subscriber*, we create a subscriber by accessing the **`subscribe()`** method of the same Observable object we created:

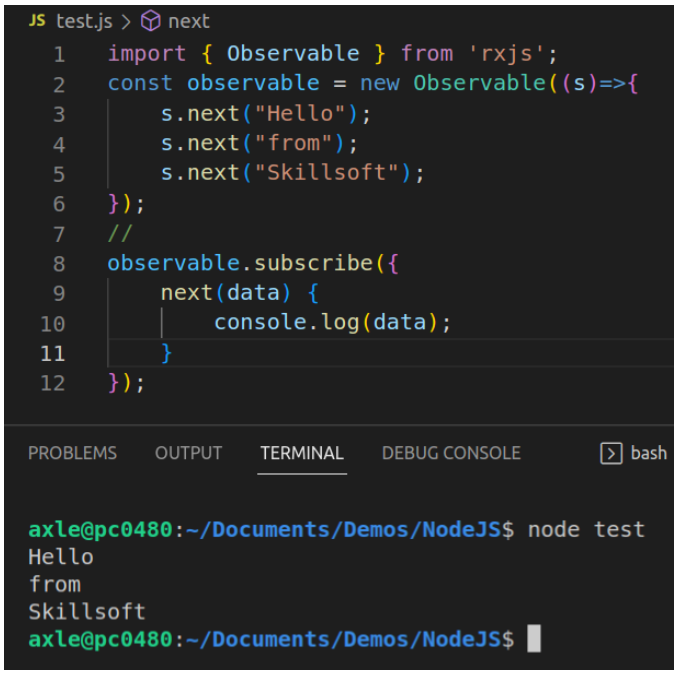
```
observable.subscribe({

});
```

Notice that the subscribe method takes as a parameter, an object. That oboect must be configured with three functions, passed in as the *object* parameters.

8. It is up to us to configure the object that is part of the subscription. The observer inside the Observable can fire off one of three methods, **`next()`**, **`error()`** and **`complete`**(). We have to have matching methods in our object that is inside of the subscribe method. Lets first configure the **`next()`** method:

```
observable.subscribe({
    next(data) {
        console.log(data);
    }
});
```

9. Here is a screenshot of VS Code showing the entire program so far and the terminal output:



Note, you must run this code in the terminal window, the browser will throw an error when it sees this code.

10. (Optional) If we are only interested in the data emitted then there is no need to configure an object in our **`subscribe()`** method, we can simply do this:

```
observable.subscribe((x)=>console.log(x));
```

11. (Optional) Showing each piece separately, so observable, observer and subscription

`import { Observable } from 'rxjs';`
`//`
`const observable = new Observable(obs =>`
`  obs.next("Hello")`
`);`
`//`
`const observer = {`
`  next: aValue => console.log("we got " + aValue)`
`}`
`//`
`observable.subscribe(observer);`
	Notice that we do not pass the const observer into the Observable constructor, pass a reference to the observer.

12. (Optional) One more method where we actually pass the *observer* object into the Observable constructor.

`import { Observable } from 'rxjs';`
`//`
`const observer = {`
`  next: aValue => console.log("we got " + aValue)`
`};`
`//`
`const observable = new Observable(observer.next("Hello"));`
	If you choose this option, the observer object must be defined **before** it is used. Also notice that we call the **next()** method on the observer that is passed in immediately.

Special note about the zipped code for Part 2 ending. The code for this part has to be *installed*. That means you must unzip the files into a folder. Using a terminal window or if you open the folder as a project in VSCode, you must run the following command: npm install

That command will bring the folder and the project back to a runnable state. This means that you will see a node_modules folder and a package_lock.json file appear in the project folder. This is normal.

Part 3 – Observables in Action

For this part we will switch to where we have the three separate pieces of code, Observable, Observer and the subscription between them. In the appendix, I will provide code that shows the other method of code construction, the one where we do the subscription implicitly. For this part, use the same code you had in Part2 but option #11.

1. We must now change the code in the Observable to reflect multiple emissions. This is as simple as adding curly braces to the parameter passed into the Observable's constructor:

```
const observable = new Observable(
  obs => {
    obs.next("Hello");
    obs.complete();
  }

);
```

Also I added the `complete()` function to the Observer parameter.

2. Also change the Observer to handle all three available methods. If you look at option 12 in Part 2, the observer was only configured with the `next()` method:

```
 const observer = {
  next: aValue => console.log("we got " + aValue),
  complete : () => console.log("done"),
  error : (err) => console.log("An error occured " + err)
}
```

3. This is the entire index.js file to this point:

```
import { Observable } from 'rxjs';
const observable = new Observable(
  obs => {
    obs.next("Hello");
    obs.complete();
  }
);
const observer = {
  next: aValue => console.log("we got " + aValue),
  complete : () => console.log("done"),
  error : (err) => console.log("An error occured " + err)
}
observable.subscribe(observer);
```

Test the code, you should see the original data plus "done".

4. Lets insert an error into the Observable via the observer:

```
const observable = new Observable(
  obs => {
    obs.next("Hello");
    obs.error("Oops!");
    obs.complete();
    obs.next("Hello");
  }
);
```

If an error did occur, depending on where the error occurred, you will get the error message but nothing else, the Observable will be finished at that point

5. Remember the Observable is best used when there is a stream of data, lets simulate that by having multiple data emissions and implementing the `setTimeout()` method again:

```
const observable = new Observable(
  obs => {
    setTimeout(()=>obs.next("hello"),1000);
    setTimeout(()=>obs.next("from"),2000);
    setTimeout(()=>obs.next("Skillsoft"),3000);
    setTimeout(()=>obs.complete(),4000);
  }
);
```

```
//using traditional functions
const observable = new
Observable((obs)=>{
    setTimeout(function(){
        obs.next("hello");
    },1000);
});
```

Remove the `error()` method.

6. One of the biggest features of observables is it's set of *operators*. For example the *of* operator can be used to create an Observable based on values you pass to it. You would have to specifically import this operator then implement it:

```
import { of } from 'rxjs';
//
const greeting = of("Hello", "from", "Skillsoft!")
```

At this point, *greeting* represents an Observable with all the features of an observable, including the data.

7. Since greeting has been converted into an Observable via the *of* operator, we can subscribe to it. At the same time I want to concatenate all the values so I create a variable to hold the final sentence:

```
let greet = "";
const greeting = of("Hello", " from ", "Skillsoft!");
greeting.subscribe(val => {
    greet += val;
});
```

8. All we have to do now is print greeting: This is all the code for this example

```
import { of } from 'rxjs';
let greet = "";
const greeting = of("Hello", " from ", "Skillsoft!");
greeting.subscribe(val => {
    greet += val;
});
console.log(greet);
```

9. (Optional) Here is a much shorter program that does the same thing

```
let greet = "";
of("Hello", " from ", "Skillsoft!").subscribe(val => {
    greet += val;
});
console.log(greet);
```

Part 4 – Removing Observables

When an Observable is created, it uses memory. In addition, each `next()` method that is fired, uses up memory and processing time. It is important therefore to unsubscribe from any observables we initially subscribed to.

You may use the same files from Part3 but remove all the code from index.js

1. We will use the *interval* class from the rxjs library to create an Observable that never stops emitting values:

```
import { interval } from 'rxjs';
//
const counter$ = interval(500);
```

2. Now if we subscribe to counter$ we could accept the values emitted by interval and log them to the console:

```
import { interval } from 'rxjs';
//
const counter$ = interval(500);
//
counter$.subscribe((x)=>console.log(x));
```

3. The problem now is that if you run this code, the numbers will just go on counting, they wont stop. This is not good, it creates a memory leak. In Linux systems you can use **CTRL-C** to stop the daemon that will be called.

4. In order to solve this problem, we need to *unsubscribe* from the Observable. But first we have to **store** the subscription in a regular variable.

```
import { interval } from 'rxjs';
//
const counter$ = interval(500);
//
let counter = counter$.subscribe((x)=>console.log(x));
```

Note, we store the subscription NOT the observable. Also `counter` is different from `counter$`

5. Once we have a handle onto the subscription, we can simply call the `unsubscribe()` method to delete the object and clean up the memory:

```
import { interval } from 'rxjs';
//
const counter$ = interval(500);
//
let counter = counter$.subscribe((x)=>console.log(x));
counter.unsubscribe();
```

Of course if you did this, you will not see anything on the screen as the `unsubscribe()` method fires immediately

6. Now, you could wrap the *unsubscribe()* inside of a *setTimeout()* method so that at least a few values can be emitted:

```
setTimeout(()=>{
    counter.unsubscribe();
}, 3000);
```

Part 5 – RxJS Operators

A very special part of the RxJS library is the collection of **operators** it has. There are several of them and they have to be studied to determine where they can be used. In this example we will use just a few. You may delete everything from you index.js file and start fresh from #1 below.

1. Add the *of* operator to your import statement:

```
import { of } from 'rxjs';
```

You may delete Observable or interval if you want or just leave it, we wont be using it.

2. Now change the Observable to the following:

```
const observable = new of(
    "Hello",
    "from",
    "Skillsoft"
);
```

3. Pass an Observer to this new Observable exactly like we did in part3 using the implicit method:

```
observable.subscribe( {
  next(data) {
      console.log(data);
  }
} );
```

4. Continue with the *error* and *complete* handlers:

```
observable.subscribe({
    next(data) {
        console.log(data);
    },
    error(err){
        console.log(err)
    },
    complete(){
        console.log("all done")
    }
});
```

As long as there is data, the **`next()`** method will fire, at the end of that data stream, the **`complete()`** method will fire.

5. (optional arrow functions)

```
observable.subscribe({
    next : data => console.log(data),
    error : err => console.log(err),
    complete : () => console.log("all done")
});
```

6. Add the *map* operator via the import statement:

```
import { of, map } from 'rxjs';
…
```

7. One of the most important operators in the library is the `pipe()` operator. It is used almost anytime we need to alter the data from an Observable in some way. Here we can use the `pipe()` operator to pass each emitted value into the `map()` operator:

```
import { of, map } from 'rxjs';
…
observable.pipe(
    map( x => x.toUpperCase())
)
```

The `map()` operator will take a projection function, in this case `toUpperCase()` and it will pass each emitted value to it. You do not have to explicitly import the `pipe()` operator.

8. As usual, you would subscribe to this Observable by chaining on the `subscribe()` method to the `pipe()` method:

```
observable.pipe(
    map( x => x.toUpperCase())
)
.subscribe( x => console.log(x))
```

9. (Optional) You could use your data from Observables in many ways, here we push into an array:

```
import { of, map } from 'rxjs';
let sentence = [];
…
//
observable.pipe(
    map(x=>x.toUpperCase())
)
.subscribe(x => sentence.push(x))
//
console.log(sentence);
```

You may now close this file, we wont be using it again

Part 6 – Async JS Library

Async is a JS utility module with about 70 easy to use, powerful functions that can be applied to asynchronous situations in JS.
This library was developed for NodeJS but can be used in browser applications.

Some of the functions include *map*, *reduce* and *filter*, some of the same kind of operations that can be found in teh RxJS library. In most of their applications you configure a single callback to handle end of operations.

Since the library is quite large, we will demonstrate only one of their available functions. In fact this is from the Control Flow part of the library.

The function we will be using is the *parallel* function. It calls other functions in parallel, without waiting until the previous function has completed. If any of the functions generate an error, the main callback is immediately called and the error object is passed to it. Once the tasks have completed, the results, if any, are sent to the final callback in array form.

1. If you are on a Node environment, install the library using npm, so `npm i async`

2. Use an existing file or create a new one in your code editor and begin with this line:

```
import async from 'async';
```

3. With *async* installed, you may now use it and invoke the `parallel()` function. Pass to that function an array of other functions to execute:

```
async.parallel([
  () => console.log('A'),
  () => console.log('B'),
  () => console.log('C')
]);
```

4. If you run the above code, all inner functions get executed and we see the normal output just like if we called each function in a different line of code. So lets add some delay to each function:

```
async.parallel([
  () => {
    setTimeout(() => {
      console.log('A')
    }, 2000);
  },
  () => {
    setTimeout(() => {
      console.log('B')
    }, 500);
  },
  () => {
    setTimeout(() => {
      console.log('C')
    }, 1000);
  }
]);
```

5. One of the reasons for using this function is to know when all the inner functions complete. For that we use a single callback function. So pass a reference to that function via the parenthesis:

```
async.parallel([
  (cb) => {
    setTimeout(() => {
      console.log('A')
    }, 2000);
  },
  (cb) => {
    setTimeout(() => {
      console.log('B')
    }, 500);
  },
  (cb) => {
    setTimeout(() => {
      console.log('C')
    }, 1000);
  }
]);
```

6. We now have to define the function, but that definition goes towards the end of the code:

```
…
      console.log('C')
    }, 1000);
  }
], () => {} );
```

7. Now just code your callback according to your needs. For most developers, it might be an email to someone or logging to a service like *log4j* or *Winston*:

```
…
      console.log('C')
    }, 1000);
  }
], (err, results)=>{
  console.log(results);
});
```

This is where you need to refer to the documentation. It states that the first parameter must be an error handler and the second will be a handler for whatever each function emits once they are done with their respective operations.

8. Once we have our callback function configured, go back to each function and call the callback and pass to it whatever details are necessary in your situation:

```
async.parallel([
  (cb) => {
    setTimeout(() => {
      console.log('A');
      cb(null, "A Completed");
    }, 2000);
  },
  (cb) => {
    setTimeout(() => {
      console.log('B');
      cb(null, "B Completed");
    }, 500);
  },
  (cb) => {
    setTimeout(() => {
      console.log('C');
      cb(null, "C Completed");
    }, 1000);
  }
], (err, results)=>{
  console.log(results);
});
```

Bonus – Observables in Angualar

Angular is a development platform that uses components to build scalable front end web applications. It is part of a group of platforms like ReactJS and VueJS. It is popular since it features lots of libraries and developer tools.

Angular apps can perform routing, handle form data, support client-server communication and lots more. Angular is also famous for its scalability, its ability to incorporate the latest developents in JavaScript and of course its ecosystem. Angular uses TypeScript, a superscript of JavaScript.

We will now create an Angular application, make an API call to the free *JsonPlaceholder* testing portal and then display the data in a browser window.

For this part you will need to have Angular installed, in a NodeJS environment. It can be installed with this command:

```
npm install -g @angular/cli
```

1. Find a suitable folder on your system (I'll be using my demos folder) and run the command to create an Angular application:

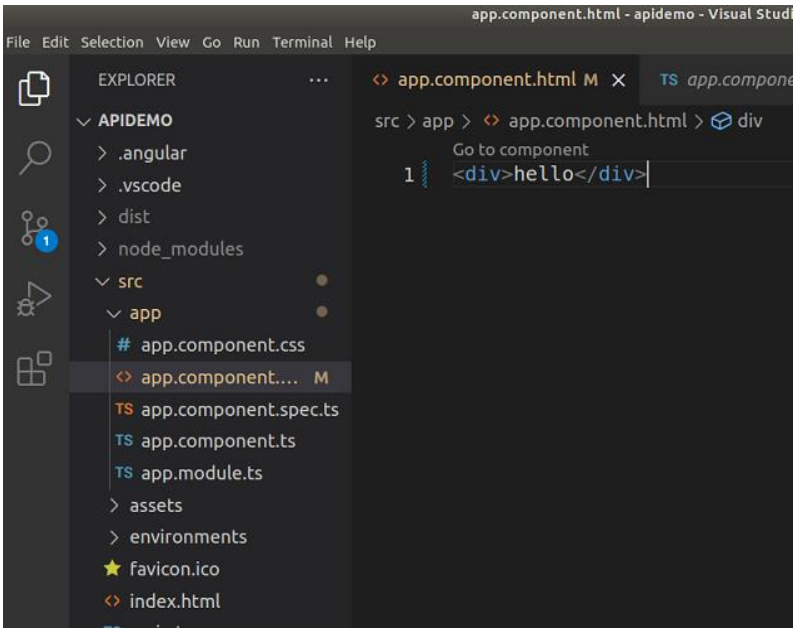
```
ng new apidemo --skip-tests --skip-git
```

You may choose N for routing and choose the default CSS option when offered

2. Now CD into the apidemo folder and run the command:

```
ng build
```

3. Open the application via it's folder in VS Code or any suitable editor. Open the app.component.html file (in src then app) and remove everything but replace it with a pair of tags, you could put some text in there just to that we see something on the web page:



4. At this point you could run the app just to see it in action. For that, go to the terminal window and type in `ng serve` or `ng serve -o`. If successful, the app will be served on localhost at port 4200.

If you are on VS Code there is a terminal window as part of the application.

5. Most of the work will be in the app.component.ts file, so open that in your editor and begin to code for the api call. We will use a method that comes with all Angular components, `ngOnInit()`

```
export class AppComponent {
  title = 'apidemo';
  //
  ngOnInit(): void {
    //api call goes here
  }
}
```

Note, this method goes inside of the AppComponent class

6. We will come back to the method soon, lets now add a constructor to the class:

```
  title = 'apidemo';
  //
  constructor() { };
  //
  ngOnInit(): void {
    //api call goes here
  }
```

7. Since we will be receiving data, we need something to put it into, we create a new variable which has to be of the Observable type. Also we don't know what kind of data will be returned so we indicate to the Observable the generic type of <any>:

```
export class AppComponent {
  title = 'apidemo';
  todo$!: Observable<any>;
  //
  constructor() { };
```

Note: the ! mark is to tell VS Code that we are certain that we will receive data. At this point there will be a red squiggly line below Observable, just hover over it and VS Code will help you resolve the issue, choose the *quick fix*.

8. We need to add a service to our application to do the heavy lifting in getting the data. So open the app.module.ts file, which is like the `main()` method of Java/C++ and configure it. Import the *HttpClientModule* from the package shown:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from "@angular/common/http";
import { AppComponent } from './app.component';


@NgModule({
```

You may choose N for routing and choose the default CSS option when offered

9. Now implement that module inside of the imports array of the same file:

```
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ]
```

You may now close this file, we wont be using it again

10. Back in the app.component.ts file, import the *HttpClient* module from the same path:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/internal/Observable';
import { HttpClient } from "@angular/common/http";
//
@Component({
  selector: 'app-root',
```

11. Inject this HttpClient module into the constructor of our *AppComponent* class via the classes constructor:

```
  todo$!: Observable<any>;
  //
  constructor(private http : HttpClient) { };
  //
  ngOnInit(): void {}
```

Note, you must include the private modifier here

12. With the service available, lets use it to make the api call and store the object being returned:

```
  constructor(private http : HttpClient) { };
  //
  ngOnInit(): void {
    this.todo$ = this.http.get(
      'https://jsonplaceholder.typicode.com/todos/1'
    );
  }
```

13. Now it's a good time to start the application, so enter the command `ng serve` into the terminal window and then open a browser pointing to http://localhost:4200. You should see just the *hello* that you entered at #3. However since our data is locked inside of `this.todo$` we need to subscribe to this Observable in order to get the data

14. For now we can simply pass the data to our one class property, title:

```
  ngOnInit(): void {
    this.todo$ = this.http.get(
      'https://jsonplaceholder.typicode.com/todos/1'
    );
    this.todo$.subscribe(data => {this.title=(data)});
  }
```

15. Now over in the html template, change the *hello* into an actual variable:

```
<div>{{this.title}}</div>
```

16. This will just show an object, so in the .ts file, you can start extracting the data like this:

```
);
  this.todo$.subscribe(data => {this.title=(data.title)});
```

Appendix A – Promisses and Loops

Here is an example of working with Promise objects coming out of a loop

1. In this example we have an array of names all in lower case. We then loop through those elements passing each name to a function called properName. The result is that we print each name after the function call and so we get all names printed using an uppercase letter for the first position in the name:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {

      let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
      return ucName;
};
//
lcNames.forEach(
      upperName => console.log(properName(upperName))
);
```

2. If we then wrap the return part of the properName() function into a setTimeout() function, then we turn that function into an asynchronous one, and must therefore treat it differently:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      setTimeout(() => {
          let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
          return ucName;
      },2000);
};
//
lcNames.forEach(
      upperName => console.log(properName(upperName))
);
```

This code will produce undefined three times

3. In order to handle this situation we have to return a Promise object and then deal with it later.

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {});
      setTimeout(() => {
          let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
          return ucName;
      },2000);
};
//
```

```
lcNames.forEach(
      upperName => console.log(properName(upperName))
);
```

Here I just bring in the Promise object, we have not used *resolve()* as yet.

4. In order to handle this situation we have to return a Promise object and then deal with it later.

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {
          setTimeout(() => {
              let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
              return ucName;
          },2000);
      });
};
//
lcNames.forEach(
      upperName => console.log(properName(upperName))
);
```

Now even the setTimeout() is wrapped inside of the Promise constructor.

5. Now that we have the Promise in place let us *resolve* the data:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {
          setTimeout(() => {
              let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
              resolve(ucName);
          },2000);
      });
};
//
lcNames.forEach(
      upperName => console.log(properName(upperName))
);
```

6. Lets now turn our attention to the forEach() loop. Right now it is just a loop chained on to the array, but if we put this into a named function we can then call that function and also make that function asynchronous:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {
          setTimeout(() => {
              let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
              resolve(ucName);
          },2000);
      });
};
```

```
//
const ucNamesArray = () => {
    lcNames.forEach(
          upperName => console.log(properName(upperName))
    );
};
```

We can now call this function anytime we want

7. So, because the properName() function returns it's data asynchronously, we can make this upperName() function *asynchronous* and therefore *await* the result from the Promise:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {
          setTimeout(() => {
              let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
              resolve(ucName);
          },2000);
      });
};
//
const ucNamesArray = async() => {
    lcNames.forEach(
          upperName => console.log(await properName(upperName))
    );
};
```

We can now call this function anytime we want

8. The above code will throw an error. Although ucNamesArray() is a function, so is the function inside of the forEach() metod. We therefore need to make that inner function async also:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {
          setTimeout(() => {
              let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
              resolve(ucName);
          },2000);
      });
};
//
const ucNamesArray = async() => {
    lcNames.forEach(
          async upperName => console.log(await properName(upperName))
    );
};
//
ucNamesArray();
```

9. All we have to do now is call the ucNamesArray() function and we get the result:

```
const lcNames = ["jones", "rosales", "rahm"];
//
const properName = lcName => {
      return new Promise( (resolve, reject) => {
          setTimeout(() => {
              let ucName = lcName.charAt(0).toUpperCase() + lcName.slice(1);
              resolve(ucName);
          },2000);
      });
};
//
const ucNamesArray = async() => {
    lcNames.forEach(
          async upperName => console.log(await properName(upperName))
    );
};
//
ucNamesArray();
```

Here is an alternative solution, one without the async/await construction.

1. We could of course, loop through each name in the array and print them out. If we did we would just have the original names, no uppercase first letter:

```
lcNames.forEach(function(eachName){
      console.log(eachName);
});
```

2. What we need to do is call the properName() function, but that returns a Promise:

```
lcNames.forEach(function(eachName){
      console.log(properName(eachName));
});
```

3. The reason is that we call log() before we receive the result of the properName() function. So we really should wait for the properName() function to finish. Since that is a process, lets start with storing the promise object into a constant:

```
lcNames.forEach(function(eachName){
      let newName = properName(eachName);
});
```

4. This makes the code less complex, because now we can chain a then() method onto the newName which is a Promise object:

```
lcNames.forEach(function(eachName){
	console.log(properName(eachName));
});
```

5. With the then() method in place, all we have to do now is pass a function into the method to handle the data being returned from properName():

```
lcNames.forEach(function(eachName){
	console.log(properName(eachName));
	newName.then();
});
```

6. Of course we have to configure the function:

```
lcNames.forEach(function(eachName){
	newName.then(function(finalName){
		console.log(finalName);
	});
});
```

7. With arrow functions:

```
lcNames.forEach(eachName => {
	let newName = properName(eachName);
	newName.then(finalName => {
		console.log(finalName);
	});
});
```

8. Without the intermediary variable

```
lcNames.forEach(eachName => {
	properName(eachName)
	.then(finalName => {
		console.log(finalName);
	});
});
```

Appendix B – Promisses Compared to Observables

A Promise handles one set of data, it emits just that data. The data can be any data type including objects but the Promise is done after the resolve or reject methods are completed.

On the other hand an Observable can handle multiple emissions. An observer will continue to receive data until the Observable encounters an error or a complete event. Observables can therefore handle streams of data.

A Promise cannot be canceled but an observer can decide to not listen anymore for values emitted by the observable. Although not a big difference, we can think of Promises as being eager, in terms of execution. On the other hand Observables are lazy. They only emit data if there is an active subscription.

Observables come with operators. This means that data can be modified before it gets to the recipient.

Appendix D – Part3 with Implicit Subscription

For this part we will switch to where we have the three separate parts, Observable, Observer and the subscription between them. In the appendix, I will provide code that shows the other method of code construction, the one where we do the subscription implicitly.

1. The Observable can emit normal values but it can also tell the subscriber when it is done and also if an error occurred somewhere in the stream of data. The subscriber method has to be configured to handle any of these three situations:

```
observable.subscribe( {
    next(data) {
        console.log(data);
    },
    complete() {
        console.log("done");
    },
    error(err) {
        console.log("An error occured " + err);
    }
} );
```

Note, there are some situations that do NOT complete for example a click on button on the DOM will never fire a *complete*. Yes, a button click is an Observable that can be subscribed to.

2. With the subscriber now prepared to handle all three methods, lets test the `complete()` function. This method does not pass data back, it is just a Boolean test on the other side. On the *subscriber* side, you can add your message as we did in #1 above:

```
    s.next("Skillsoft");
    s.complete();
});
//
observable.subscribe( {
    next(data) {
        console.log(data);
    },
    complete() {
        console.log("done");
    },
```

3. If an error did occur, depending on where the error occurred, you will get the error message but nothing else, the Observable will be finished at that point:

```
const observable = new Observable((s)=>{
    s.next("Hello");
    s.next("from");
    s.error("communication lost");
    s.next("Skillsoft");
    s.complete( );
});
```

4. A more compact way to design your `subscribe()` method is to use *name : value* pairs, with the value side being a function:

```
cobservable.subscribe( {
    next : (data) => console.log(data),
    complete : () => console.log("done"),
    error : (err) => console.log("An error occured " + err)
} );
```

5. Remember the Observable is best used when there is a stream of data, lets simulate that by implementing the `setTimeout()` method again:

```
const observable = new Observable((s)=>{
    setTimeout(()=>s.next("hello"),1000);
    setTimeout(()=>s.next("from"),2000);
    setTimeout(()=>s.next("Skillsoft"),3000);
    setTimeout(()=>s.complete(),4000);
});
```

```
//using traditional functions
const observable = new
Observable((s)=>{
    setTimeout(function(){
        s.next("hello");
    },1000);
});
```

Remember the Observable is best used when there is a stream of data, lets simulate that by implementing the `setTimeout()` method again:

6. One of the biggest features of observables is it's set of operators. For example the *of* operator can be used to create an Observable based on values you pass to it. You would have to specifically import this operator then implement it:

```
import { of } from 'rxjs';
//
const greeting = of("Hello", "from", "Skillsoft!")
```

   At this point, *greeting* represents an Observable with all the features of an observable, including the data.

7. Since greeting has been converted into an Observable via the *of* operator, we can subscribe to it. At the same time I want to concatenate all the values so I create a variable to hold the final sentence:

```
let greet = "";
const greeting = of("Hello", " from ", "Skillsoft!");
greeting.subscribe(val => {
    greet += val;
});
```

8. All we have to do now is print greeting: This is all the code for this example

```
import { of } from 'rxjs';
let greet = "";
const greeting = of("Hello", " from ", "Skillsoft!");
greeting.subscribe(val => {
    greet += val;
});
console.log(greet);
```

9. (Optional) Here is a much shorter program that does the same thing

```
let greet = "";
of("Hello", " from ", "Skillsoft!").subscribe(val => {
    greet += val;
});
console.log(greet);
```