RxJS & Angular

Learning Objectives

- Know which parts of Angular expose observables.
- Know how to *practically* use RxJS with Angular Forms.
- Know how to chain common RxJS operators together.
- Know that you don't need to use observables in Angular and how to implement the same solution without an observable chain.

Angular observables

There are a few places in Angular where reactive programming and observables are in use.

EventEmitter

Under the hood this works via Observables.

HTTP

We've not covered this yet but HTTP requests in Angular are all handled via Observables.

Forms

Reactive forms in Angular expose an observable, a stream of all the input fields in the form combined.

In this lecture we'll use Forms as an example of how to do Reactive Programming in Angular.

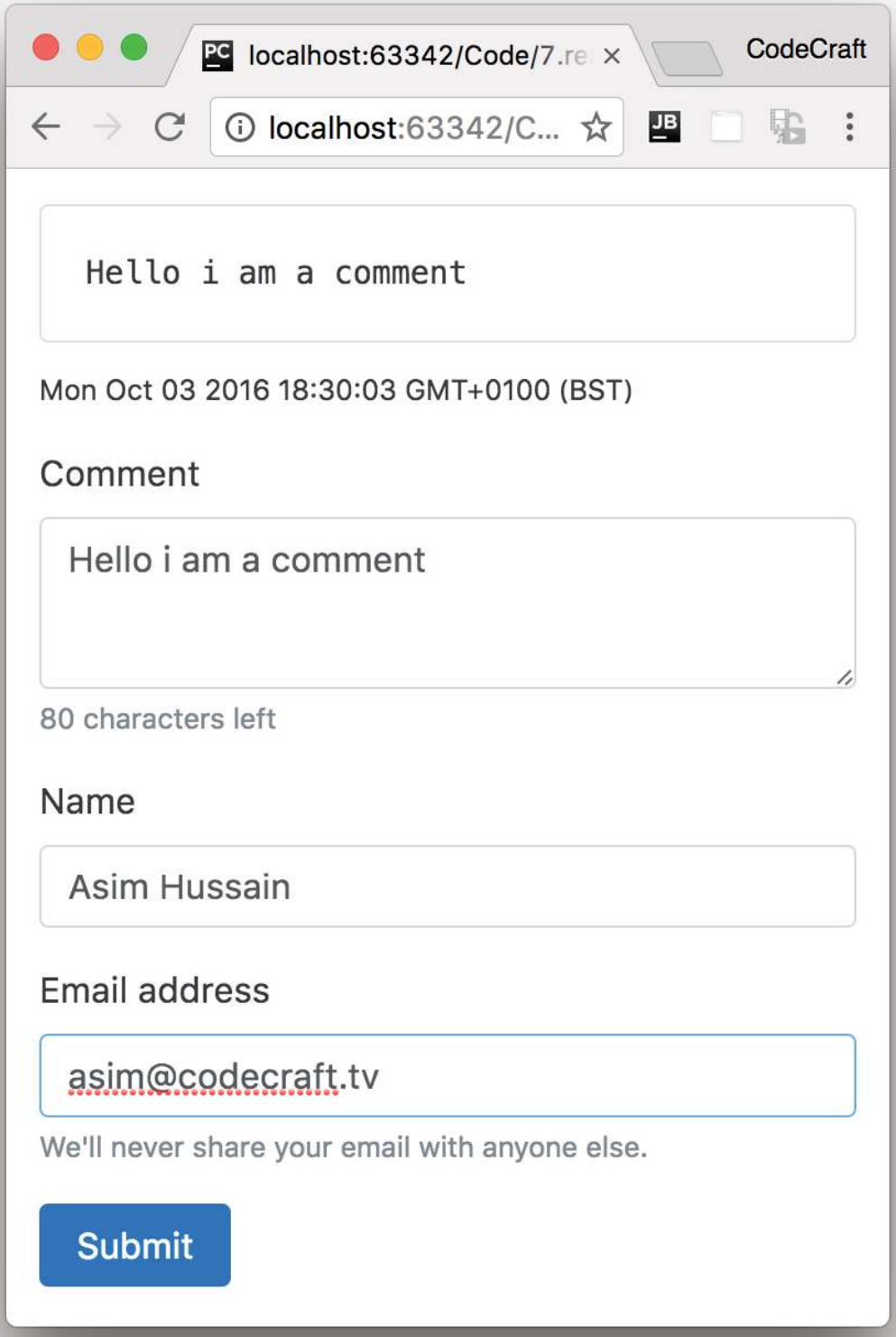
In order to teach you a practical example of RxJS in Angular I have to jump ahead and use something we have not covered yet, *Forms* and specifically the *Reactive Forms* module.

This is not the only way to implement forms in Angular. This is the reactive *model driven* approach, there is also the template driven approach, will go through both in the course.

We'll cover forms in *much* more detail later on in this course so for now I'm going to *gloss* over a lot of the details and focus *only* on the code that deals with RxJS.

Reactive form example

In our example we have a `FormAppComponent` which renders a form like so:



We have a comment input area, a name and email input field and also a submit button.

We won't go through the listing for the template HTML, since it's complex and won't make sense

until we cover forms later on, but we will go through the code for the component class, like so:

form-app-component.ts

```
import 'rxjs/Rx'; ①
.
.
.
class FormAppComponent {
  form: FormGroup; ②
  comment = new FormControl("", Validators.required); ③
  name = new FormControl("", Validators.required); ③
  email = new FormControl("", [ ③
    Validators.required,
    Validators.pattern("[^ @]*@[^ @]*")
  ]);

  constructor(fb: FormBuilder) {
    this.form = fb.group({ ④
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
  }

  onSubmit() {
    console.log("Form submitted!");
  }
}
```

① We import the *full* RxJS library into our application (this import includes *all* the operators).

② We create a `form` property which will hold a representation of our form so we can interact with it from code.

③ We create individual instances of controls and rules for when user input is valid or invalid.

④ We then link our `form` with the controls we created in the constructor using something called a `FormBuilder`

We create a `form` instance on our component, this instance exposes an observable, a stream of all the input fields combined into a object, via it's `valueChanges` property.

We can subscribe to that observable and print our the current value of the form, like so:

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

Then as I type into the form, this below gets printed to the console.

```
{"comment":"f","name":"","email":""}
{"comment":"fo","name":"","email":""}
{"comment":"foo","name":"","email":""}
{"comment":"foo","name":"a","email":""}
{"comment":"foo","name":"as","email":""}
{"comment":"foo","name":"asi","email":""}
{"comment":"foo","name":"asim","email":""}
{"comment":"foo","name":"asim","email":"a"}
{"comment":"foo","name":"asim","email":"as"}
{"comment":"foo","name":"asim","email":"asi"}
{"comment":"foo","name":"asim","email":"asim"}
{"comment":"foo","name":"asim","email":"asim@"}
```

Processing only valid form values

Looking at the stream above we can see that *most* of the stream items are for invalid forms; comment, name and email are required so any form without a value for those is invalid. Also we have some special validation logic for the email field, it's only valid if it contains an @ character.

In fact the only valid stream item is the last one `{"comment":"foo","name":"asim","email":"asim@"}`.

That's a common issue when dealing with forms, we only want to *bother* processing the results of a *valid* form, there really isn't any point processing invalid form entries.

We can solve this by using another RxJS operator called `filter`. `filter` accepts a function and passes to it each item in the stream, if the function returns true `filter` publishes the input item to the output stream.

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

`this.form.valid` is true when the whole form is valid. So while the form is *invalid* `.filter(data ⇒ this.form.valid)` *doesn't* push items to the output stream, when the form is *valid* it does start pushing items to the output stream.

The end result of the above is that when we type into the form the same data as before, the only item that gets published in our subscribe callback is:

```
{"comment":"foo","name":"Asim","email":"asim@"}
```

Cleaning form data

A comment input box is a dangerous place, hackers try to input things like `<script>` tags and if we are not careful we open ourselves to the possibility of hackers gaming our applications.

So one common safety measure for comment forms is to strip our app html tags from the message before we post it anywhere.

We can solve that again via. a simple `map` operator on our form processing stream.

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .map(data => {
          data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
          return data
        })
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

We add another operator to our stream, specifically we added a `map` operator.

We added this *after* the `filter` operator, so this `map` operator only gets called when the previous `filter` operator publishes to it's output stream. To put it another way, this map operator only gets called on *valid* form values.

```
.map(data => { ①
  data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, ''); ②
  return data ③
})
```

① The `map` operator gets passed the form object as a parameter called `data`.

② We apply a regular expression on the comment property to replace everything that could be a HTML tag with an empty string.

③ What we return from the map function is what gets pushed to the `map` operators output stream.

Now when we type into our form a comment of `<script>` this is what gets printed out:

```
{"comment":"<","name":"Asim","email":"asim@"}
{"comment":"<s","name":"Asim","email":"asim@"}
{"comment":"<sc","name":"Asim","email":"asim@"}
{"comment":"<scr","name":"Asim","email":"asim@"}
{"comment":"<scri","name":"Asim","email":"asim@"}
{"comment":"<scrip","name":"Asim","email":"asim@"}
{"comment":"<script","name":"Asim","email":"asim@"}
{"comment":"","name":"Asim","email":"asim@"}
```

Focusing on the last line `{"comment":"","name":"Asim","email":"asim@"}` we can see that the `<script>` tag the user typed in, is stripped from the comment property.

Adding form values

A useful feature would be if we could let the user know the last time the form was updated.

We can solve this again by adding another map operator to our observable chain, this time we just add the current time to another property to our data object called `lastUpdateTS`, like so:

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .filter(data => this.form.valid)
        .map(data => {
          data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
          return data
        })
        .map(data => {
          data.lastUpdateTS = new Date();
          return data
        })
        .subscribe( data => console.log(JSON.stringify(data)));
  }
```

This second map operator simply adds a property to our data object and then pushes the data object onto its output stream.

If we now ran our application we would see this printed out:

```
{"comment":"f","name":"Asim","email":"asim@","lastUpdateTS":"2016-10-
03T20:33:45.980Z"}
{"comment":"fo","name":"Asim","email":"asim@","lastUpdateTS":"2016-10-
03T20:33:46.187Z"}
{"comment":"foo","name":"Asim","email":"asim@","lastUpdateTS":"2016-10-
03T20:33:46.364Z"}
```

Not using observables

In Angular we don't *need* to use observables, and therefore reactive programming, if we don't want to.

Instead of adding operators to the observable chain we can choose to *just* subscribe and do the processing in the callback, like so:

```
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "comment": this.comment,
      "name": this.name,
      "email": this.email
    });
    this.form.valueChanges
        .subscribe( data => {
          if (this.form.valid) {
            data.comment = data.comment.replace(/<(?:.|\n)*?>/gm, '');
            data.lastUpdateTS = new Date();
            console.log(JSON.stringify(data))
          }
        });
  }
```

In the above example there doesn't seem to be much advantage to reactive programming via an observable chain vs. just coding up as you are used to in the subscribe callback.

The advantage of RxJS and Observables come to play when we start using more of the complex operators like `debounce` and `distinctUntilChanged`. Implementing the same functionality as those operators via standard imperative coding techniques would take many more lines of code than the equivalent RxJS solution.

Summary

Angular exposes RxJS observables in a small but important number of places in Angular. The EventEmitter, HTTP and Reactive Forms.

We use `operators` to add to the observable chain and then subscribe to the output and perform *actual* real life actions in our application, either change the state of variables or call functions.

We can choose to take advantage of that and code reactively, or we can just subscribe to the observable and code imperatively.

Listing

http://plnkr.co/edit/FbBMirJsnerQqnUXGZel?p=preview

script.ts

```
import {NgModule, Component} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {ReactiveFormsModule, FormGroup, FormControl, Validators, FormBuilder} from
"@angular/forms";
import 'rxjs/Rx';
```