Dr Tomasz Kurowski
t.j.kurowski@cranfield.ac.uk

Optional: Web development using Angular

In this practical you will revisit and expand on a previous JavaScript task from the Data Integration module. You will use a Web development framework called Angular to build web applications. Angular uses the TypeScript programming language, an expanded version of JavaScript which has a more complex (static) type system and – especially as used by Angular – leverages many advanced JavaScript features such as a class and module system (those are available, but optional, in "plain" JavaScript as well). This TypeScript code, organized according to the principles of the Angular framework, will ultimately be compiled (translated) into "plain" JavaScript, ready to be used by a browser, when we build our Angular application.

The Angular ecosystem includes a command line interface tool (Angular CLI) which makes it easier to set up and organise your application. Let's install it first:

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

Note that this is a global (`-g`) installation, not part of any specific project. Angular CLI is used via the `ng` command, for example:

```
ng help
```

You can create an Angular project with the following command:

```
ng new first-practical --no-standalone
```

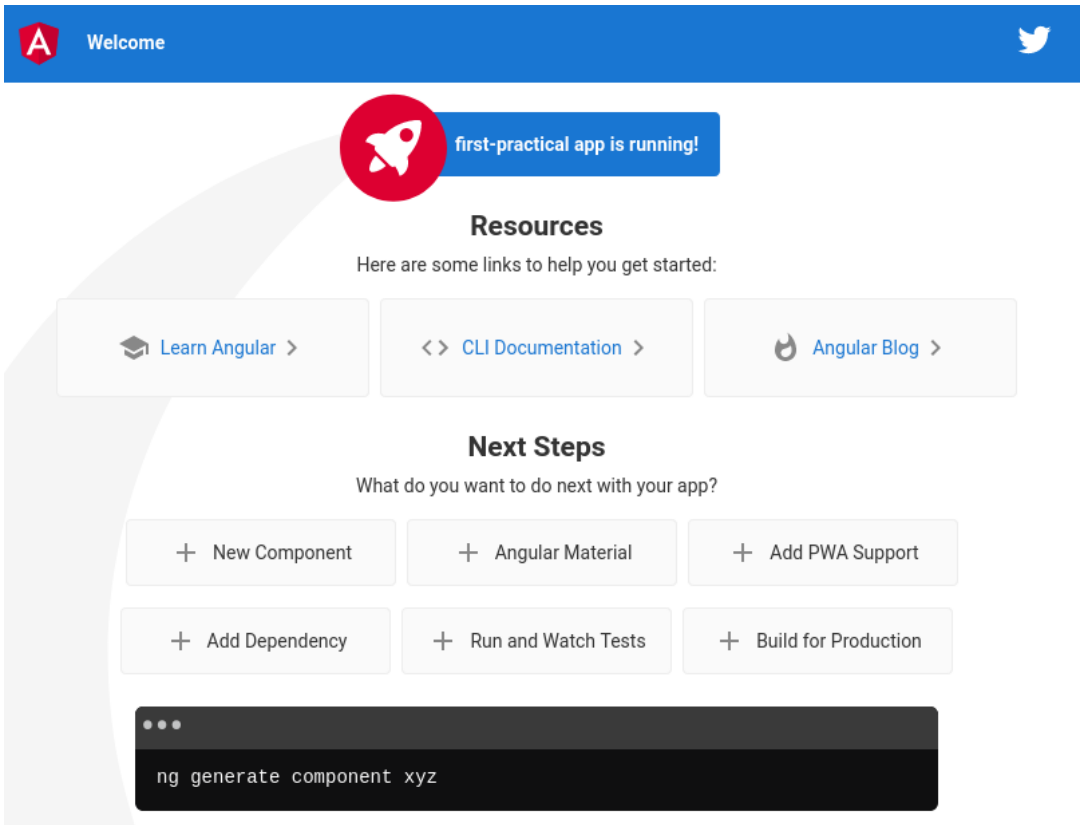
Use default options for the initial setup. Once it is complete, have a look at the contents of the newly created project directory. Note that a `package.json` file has been created for you (and Angular with its dependencies is already

installed in `node_modules`), and a Git repository with an initial commit has been set up as well (including a `.git` directory and a `.gitignore` file). There is also an `angular.json` configuration file for the Angular settings of the project, as well as files for TypeScript compilation and linting settings, and likely a configuration file for a testing tool called Karma. We will be ignoring most of those to start with – the default settings are good enough for us.

You can build and run your new project by entering its main directory and using the following command:

```
ng serve
```

The above will deploy your Web application on a development server running on http://localhost:4200/ and continue running until interrupted (e.g. by pressing Ctrl+C). It will also automatically update as you modify your code, so you can leave it running while you work. Have a look at your app in a browser.



Take a look at the project directory using the environment of your choice (I recommend Visual Studio Code). Most of the code of your application is inside the `src` directory. Take a look at the `index.html` file, which forms the entry point of your Web application. The page body should look as follows:

```
<body>
  <app-root></app-root>
</body>
```

Now, `<app-root>` is not a standard HTML tag. It represents an Angular **component**. Components are the building blocks of Angular applications, representing parts of a page with their own UI and functionality. A component is generally composed of three parts, which for our root component is located at `src/app`:

- A component class (in `app.component.ts`) – this will contain your component functionalities as well as component metadata. The latter is stored in the `@Component` decorator, and includes the selector (`"app-root"`, the name of our component tag) and paths to the files containing the other two parts.
- An HTML template (in `app.component.html`) – this will contain the HTML structure of your component and use the functionalities from the component class.
- Component styles (in `app.component.css`) – this will contain the visual styles used by the component (as CSS or related format files).

At the moment both the CSS styles and HTML structure for the welcome page are contained in the `app.component.html` file. We will be replacing that with our own code.

Components have to be declared inside **modules**, which can then be imported into other modules (with at least one being bootstapped at the root of the

application). Have a look at the `app.module.ts` file, where our `AppComponent` component class is declared inside a `@NgModule` class decorator.

Let us start with a simple example. Notice that our `AppComponent` class has a string member called `title`. Change its name to `greeting` and contents from '`first-practical`' to '`Hello world!`'. Then open the app.component.html template file, and replace its contents with the code below:

```
<h2> {{ greeting }} </h2>
```

The `<h2>` is a standard HTML header tag, while the `{{` double braces `}}` are used for **interpolation binding** – TypeScript code between the braces will be executed, with the values of class properties (or return values of getters) being displayed – in this case the contents of the `greeting` property (i.e. "Hello world!") should be displayed inside of a header. Have a look!

We can also bind component class members to HTML tag properties and events. Add a `greetingCounter` property and create the following method in the component class:

```
greetingCounter = 1;
greet() {
  this.greetingCounter++;
  this.greeting = 'Hello world for the ' +
                  this.greetingCounter + ' time!'
}
```

The method above will update the greeting and increase the greeting counter by one every time it is triggered. Let's give the user a way to trigger it by adding a button with the method bound to its `onClick` event to our HTML template:

```
<button (click)="greet()">Greet</button>
```

Try it out in the browser!

If you wish you can also try modifying the method to use the correct ordinal suffix (i.e. *st*, *nd*, *rd*, *th*, as appropriate) – you did this in one of the Java module practicals!

Guestbook

Let's replicate another of the Data Integration practicals – a simple guestbooka pplication. Close the previous one and create a new Angular application (e.g.`ng new guestbook --no-standalone`). This time, instead of adding our code direc tly to the rootcomponent, we will create our own Guestbook component:

```
ng generate component guestbook
```

The above will create an empty component with a `GuestbookComponent` class and an `app-guestbook` selector. Replace the contents of your root component template (`app.component.html`) with our guestbook tag:

```
<app-guestbook></app-guestbook>
```

You should be greeted with the contents of our guestbook component, i.e. a paragraph saying "guestbook works!" when you launch the application via `ng serve`. Let us create an interface which will describe what makes up one of our messages. Create a `message.ts` file inside `src/app` containing the following:

```
export interface Message {
    author: string;
    message: string;
}
```

This means that a Message is an object containing an author string, and a message string. You can import it in our `guestbook.component.ts` as follows:

```
import { Message } from '../message';
```

Now add a `messages` property to our `GuestbookComponent`. The property will be used to store an array (designated by square brackets and initially empty) of `Message` objects:

```
messages: Message[] = [];
```

Notice that alongside a class constructor, there is a currently empty `ngOnInit` method in our class, this is an Angular **lifecycle hook**, a method which is triggered when an Angular component is initialised (there exist other similar hooks, like `ngOnChanges`, or `ngOnDestroy`). Let's manually add a couple of messages to our array inside that method:

```
ngOnInit(): void {
  this.messages.push({
    author: 'Tom',
    message: 'This is the first message in our app!'
  });
  this.messages.push({
    author: 'Tom',
    message: 'This is another message!'
  });
}
```

Time to display our messages. Add the following code to the HTML template of our guestbook component:

```
<div *ngFor="let message of messages">
  <b>{{ message.author }}</b> says: {{ message.message }}
</div>
```

The `<div>` and `<b>` tags are standard HTML, while `*ngFor` is an Angular-specific structural directive which will render its parent tag once for each item in a collection – in this case for each `message` in our `messages` array. Try it!

Tom says: This is the first message in our app!
Tom says: This is another message!

We will now add a form which will allow a user to add new messages. In order for Angular bindings to work with forms, we need to import the `FormsModule` module. Add it to the imports array in the `@NgModule` decorator in `app.module.ts`:

```
import { FormsModule } from '@angular/forms';
```

Next, add variables which will store the new message and its author, as well as a method which will add them to our list of messages, to the GuestbookComponent class:

```
currentAuthor = 'Tom';
currentMessage = '';

sendMessage() {
  if (this.currentAuthor != '' && this.currentMessage != '') {
    this.messages.push({
      author: this.currentAuthor,
      message: this.currentMessage
    });
    this.currentMessage = '';
  }
}
```

Finally, add a form to our guestbook component HTML template:

```
<div>Name:</div>
<input type="text" [(ngModel)]="currentAuthor" />
<div>Message:</div>
<input type="text" [(ngModel)]="currentMessage" />
<button (click)="sendMessage()">Send</button>
```

The `[(ngModel)]` is a **two-way binding** (implemented in `FormsModule`) which connects the contents of the two input boxes with the `currentAuthor` and `currentMessage` variables, respectively. Whenever one of them changes, the other will also be updated. The `sendMessage()` method is bound to the onClick event of the button. Try it out! You should be able to add messages.

Services

Instead of storing data directly inside components, Angular applications generally use Angular Services to either store or (more often) retrieve data from databases or external services. I have set up a shared message service (based on the one you implemented during the Data Integration module for server-side message storage), which is available at http://elvis.misc.cranfield.ac.uk/practical_api/messages. Take a look at the messages which are already there.

Let us create a service which will send and fetch data to and from our Elvis server:

```
ng generate service messages
```

A new file called `messages.service.ts` will be created for you. It contains an empty `MessagesService` class. We will use it to contact the HTTP Elvis server by using the `HttpClient` class. We need to import its module inside of the app.modules.ts file and add it to the imports:

```
import { HttpClientModule } from '@angular/common/http';
```

We also need to add it to the Service class:

```
import { HttpClient } from '@angular/common/http';
```

And inject it into the service constructor:

```
constructor(private readonly http: HttpClient) {}
```

Once injected, it can be used inside of service methods. The following two methods will retrieve (via a GET request, receiving an array of messages) and send (via a POST request, sending a single messages) to the service on Elvis:

```
getMessages() {
  let query = 'http://elvis.misc.cranfield.ac.uk/practical_api/messages';
  return this.http.get<Message[]>(query);
}

sendMessage(message: Message) {
  let query =
'http://elvis.misc.cranfield.ac.uk/practical_api/sendmessage';
  const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };
  return this.http.post<Message>(query, message, httpOptions);
}
```

Note the message header used, indicating that the contents of our POST body are in the JSON format.

To use the service from within out guestbook component, you need to inject it into the constructor of the component class:

```
constructor(private readonly messagesService: MessagesService) { }
```

The service will now be available in all of the component class methods. The `HttpClient.get` and `HttpClient.post` methods do not return `Message` objects directly, but return `Observables`, which have a `subscribe` method. The subscribe method allows you to provide a callback function which will handle the result of the request. You can implement a function which fetches messages from the service as follows:

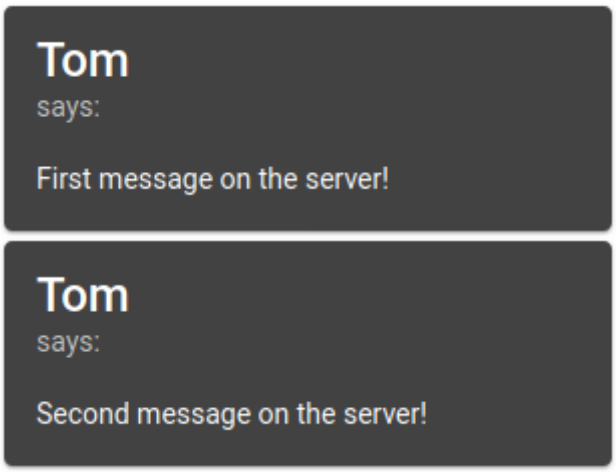
```
updateMessages() {
  this.messagesService.getMessages().subscribe(messages => {
    this.messages = messages;
  });
}
```

The callback provided in the subscribe method call is a **fat arrow** anonymous function, which we discussed briefly in the Data Integration module. The `messages` array which we receive from the Observable gets assigned to our guestbook's local `this.messages` property, which will automatically update our HTML page when reassigned.

Instead of manually populating the `messages` array inside `ngOnInit`, we can now simply call the updateMessages method:

```
ngOnInit(): void {
  this.updateMessages();
}
```

Try it! You should be able to retrieve messages from the server.



Now that you can receive messages from the server, the only thing left to do is to send now ones via a POST request and update the guestbook once the request completes. Modify your sendMessage method to use the messagesService:

```
sendMessage() {
  this.messagesService.sendMessage({
    author: this.currentAuthor,
    message: this.currentMessage
  }).subscribe(message => {
    // POST request completed, updating messages
    this.updateMessages();
  });
}
```

With the above method, you should be able to send messages to the server! Since the server and its messages are shared by all of you, you can now communicate with the other students! Try it.

Component libraries

Instead of using basic HTML components, you may prefer to use one of several existing **component libraries**, which provide ready-made components with a consistent, well-designed look-and-feel, usually with the option of selecting specific themes to enhance the interface. Popular libraries include **ngx-bootstrap**, **Prime NG**, and Google's own **Material Design** suite. Let us install the last one and make our guestbook a bit prettier:

```
ng add @angular/material
```

You can view available components, documentation and examples of their usage at https://material.angular.io/components/.

We will use the MatCard component to represent our messages, replacing simple unstyled `<div>` tags. Import their parent module inside `app.module.ts` and include it in the `imports` array of the module decorator:

```
import { MatCardModule } from '@angular/material/card';
```

You can now replace your previous `<div>` in your guestbook component HTML template with the following code:

```
<mat-card *ngFor="let message of messages">
  <mat-card-title>{{ message.author }}</mat-card-title>
  <mat-card-subtitle>says:</mat-card-subtitle>
  <mat-card-content>{{ message.message }}</mat-card-content>
</mat-card>
```

Try it, then try to find an appropriate Material Design components to use in your new message form!

Git

Solutions are available at the following Git repositories:

https://bitbucket.org/tomkurowski/angularfirstpractical

https://bitbucket.org/tomkurowski/angularguestbook

Create an online repository for your own project – if you selected the default options when setting up your Angular project (and have Git installed on your system), you should already have a local Git repository.

Extra challenge

Deploy your microbial growth API from the data integration module on http://localhost:3000 and create an Angular application which fetches data from it using an Angular Service and displays it in a table (try the Material table component: https://material.angular.io/components/table/overview).