JOSHUA MORONY

BUILDING MOBILE APPS WITH IONIC & ANGULAR

Lesson 4: Creating an Interface and Checklist Service

We have the majority of our template for the application set up so far, but we are also going to need to handle the "logic" that happens in the application. We have a page to display our checklists (which don't exist yet), and a page to display the details/items for a particular checklist, but now we need to start working on how to actually make things *happen* on those pages.

We will need a way to create checklists, create items to add to the checklists, and we will need a way to modify (update and delete) that data. This is what we will be focusing on in this lesson.

Whenever you have some kind of "entity" in your application like articles, users, products, or in this case checklists, it's generally a good idea to create a **service** that handles all of the operations for that entity. In this case, we would use our service to create new checklists, to add items to the checklist, to update a checklist, and so on. A service is not all that different to a regular page/component, but its purpose is purely to run logic - it doesn't have a template to display things directly to the user. Unlike a page/component that becomes "active" when we go to a certain route, a service is just a "helper" that is used inside of other pages/components. We can even use the same service in multiple different pages to access the same set of functionality or share data between pages.

We don't *need* to create services to handle all of the "logic" in our application, because each page that we have also has its own class that we can create functions in. We could, for example, handle all of the logic for creating and updating checklists inside of our **home.page.ts** file. However, it's best to get pages to do as little work as possible. A page

should only contain "logic" that is specific to *only* that page. For example, if we have a `(click)` listener attached to a button in the home page's template, then it makes sense to create a button and some logic for that in the **home.page.ts** file. However, any kind of complex logic, or logic that relates to "entities" in your application, should generally happen inside of a "service" that is injected into the page.

Services may also be referred to as "providers" or "injectables". We use the `@Injectable` decorator to create a service with Angular, and the service that we create can be "injected" into any of the other pages/components in our application through the constructor, e.g:

```
import { MyService } from '../../services/myservice.service';

...snip...

constructor(private myService: MyService){

}
```

After injecting `MyService` as `myService` in the example above, we would then be able to access the functionality that `MyService` provides anywhere within that class like this:

```
this.myService.someFunction();
```

Create an Interface

Before we create the service that will handle everything related to checklists in our application, we are first going to create something called an **interface**.

A **service** will allow us to handle operations related to checklists in our application, and an **interface** will allow us to define what a checklist *is*.

We've talked previously about using "types" in our Ionic applications, e.g:

```
let myString: string = "hello";
```

In this example, we have given `myString` a type of `string`. Although this type is not strictly required for the operation of the application, TypeScript will now warn us if we ever attempt to set `myString` to anything that is not a string.

Now, let's say that a checklist in our application would be an object that looks like this:

```
let myChecklist = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      title: 'Item One',
      checked: false
    },
    {
```

```
      title: 'Item Two',
      checked: true
    }
  ]
};
```

Seeing the value of TypeScript and types, we may consider this fantastic object of ours and decide: "I shall give this an appropriate type!"

We could just give it a type of `Object`, because it is an object:

```
let myChecklist: Object = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      title: 'Item One',
      checked: false
    },
    {
      title: 'Item Two',
      checked: true
    }
  ]
};
```

However, an `Object` is a pretty generic description for something that may follow a very strict format. In this case, our checklists are always going to have an `id` that is a string, a `title` that is a string, and a `items` property that will be an `array`. But a type of `Object` will only enforce that it is any kind of object.

To enforce this structure, we can create our own custom type with an **interface**.

This isn't strictly required, in fact, you don't *have* to give types to your data at all, but it's a good habit to get into. Adding appropriate types will keep the TypeScript compiler happy, and it will allow you to catch a lot of bugs/issues in your application before they ever become issues.

> Modify `src/app/interfaces/checklist.ts` to reflect the following:

```
export interface Checklist {
    id: string;
    title: string;
    items: ChecklistItem[];
}

export interface ChecklistItem {
    title: string;
    checked: boolean;
}
```

To create an interface, all we need to do is supply each property the object expects, and we need to give each property the appropriate type. Notice that for the `items` property of the `Checklist` interface that we give it a type of `ChecklistItem[]` which means "an array of elements that have a type of ChecklistItem". This `ChecklistItem` type is another interface we have defined, which describes the items that will be added to the checklist.

Later, when we are working with our checklists, we will be able to import these interfaces and use them as our own custom types:

```
let myChecklist: Checklist = {
  id: 'my-checklist',
  title: 'My Checklist',
  items: [
    {
      title: 'Item One',
      checked: false
    },
    {
      title: 'Item Two',
      checked: true
    }
  ]
};
```

The more specific you get with your types the better... but I wouldn't blame you for just

going with the more generic `Object` type here to save some time. You can even use the `any` type to completely circumvent needing a type at all - this is also the most useless type, though, because it will allow absolutely any kind of data.

Some people probably wouldn't be too happy at me suggesting that, but I think it's fine to give yourself a bit of breathing room as a beginner and not worry too much with finicky best practice type things. If you are working on a serious production application, then I would implore you to take the time to set up interfaces like this. If you are just messing around and trying to learn, I don't think it's that important.

Create the Checklist Service

Now we are going to create our checklist service. Remember, we will be injecting this service into the pages on our application so that we can control operations related to the addition, updating, or deletion of checklists and their items.

Let's start off by setting up a bit of a "skeleton" version of our service. After this, we will work through adding each of the functions in more detail.

> Modify `src/app/services/checklist-data.service.ts` to reflect the following:

```
import { Injectable } from '@angular/core';
import { Checklist } from '../interfaces/checklist';

@Injectable({
```

```
  providedIn: 'root'
})
export class ChecklistDataService {

  public checklists: Checklist[] = [];
  public loaded: boolean = false;

  constructor(){

  }

  load(): Promise<boolean> {
    return Promise.resolve(true);
  }

  createChecklist(data): void {

  }

  renameChecklist(checklist, data): void {

  }

  removeChecklist(checklist): void {

  }

  getChecklist(id): Checklist {
```

```
    return {
      id: '',
      title: '',
      items: []
    };
  }

  addItem(checklistId, data): void {

  }

  removeItem(checklist, item): void {

  }

  renameItem(item, data): void {

  }

  toggleItem(item): void {

  }

  save(): void {

  }

  generateSlug(title): string {
```

```
    return '';
  }

}
```

We have a basic service/provider/injectable set up here with a bunch of empty functions, and a class member of `checklists`. Notice that we have imported our interface and we are using this as the type for our class member:

```
public checklists: Checklist[] = [];
```

This variable will be an array of checklists. This array will contain all of the checklists that are currently available in the application. Notice that we have made this member variable `public`. We will be directly accessing the data contained in this service to display it in the pages in our application.

Each of the functions in this service will allow us to perform some kind of operation on our checklists or their items, and we will be stepping through implementing those one-by-one. Since some of these functions are set to return a particular type of data, we are returning dummy data for any of the functions that aren't `void` (i.e. they are not supposed to return anything).

BEST PRACTICE ALERT! I feel the need to acknowledge a couple of things about this service that could be better. First of all, "checklists" and "checklist items" could be

considered two different entities, and so each should probably have their own service. The `generateSlug` function is also really just a generic function and not related specifically to checklists, so it would make sense to separate that out into some kind of "utilities" service or class. However, I've intentionally left these optimisations out for the sake of making this walkthrough easier to follow.

My general approach is to keep "best practice" in mind, but don't freak out over it. You can optimise things to the nth degree, and sometimes you don't get any real benefit out of it. Again, if you are working on a huge production application with a team of developers, you should definitely worry about this stuff. If you are building a small application by yourself... probably not as important. Especially as a beginner, if you get stuff like this wrong, or you aren't putting stuff in the "best" place, try not to worry too much.

load & save

The `load` function will be responsible for intialising the `checklists` member variable with data loaded in from storage. Similarly, the `save` function will be responsible for saving data to storage. However, we will be implementing this in another lesson as we will be talking about data storage in more depth.

createChecklist

This function will be responsible for the creation of a new checklist.

> Modify the `createChecklist` function to reflect the following:

```
  createChecklist(data): void {

    this.checklists.push({
      id: this.generateSlug(data.name),
      title: data.name,
      items: []
    });

    this.save();

  }
```

It will accept some data as a parameter, and it will push a new checklist to the `checklists` array. Notice that we call the `generateSlug` function to generate the `id`. We will implement this function in a moment, and its responsibility is basically just to turn the name of the checklist into a unique `id`. Using a sensical `id` like `my-checklist` rather than an `id` like `0938242` makes for more human-readable URLs.

Finally, we call `this.save()` to trigger the saving of the data. Right now, this save call will do nothing.

renameChecklist

Next, we're going to define the `renameChecklist` function which, obviously, will allow us to rename a checklist.

> Modify the `renameChecklist` function to reflect the following:

```
renameChecklist(checklist, data): void {

  let index = this.checklists.indexOf(checklist);

  if(index > -1){
    this.checklists[index].title = data.name;
    this.save();
  }

}
```

This takes in a particular `checklist` as a parameter, as well as the `data` we want to update it with. We first find the particular checklist in our `checklists` array, and if it is found, we updated the title of that checklist with the data that was passed in.

removeChecklist

Next up we are going to add the ability to delete a checklist.

> Modify the `removeChecklist` function to reflect the following:

```
removeChecklist(checklist): void {
```

```
    let index = this.checklists.indexOf(checklist);

    if(index > -1){
      this.checklists.splice(index, 1);
      this.save();
    }

  }
```

This is quite similar to the `renameChecklist` function, except that we don't need to update it with any data. Instead of updating it, we just remove it from the array with `splice` and then trigger a save.

getChecklist

The `getChecklist` function is going to allow us to grab a particular checklist using its `id`.

> Modify the `getChecklist` function to reflect the following:

```
  getChecklist(id): Checklist {
    return this.checklists.find(checklist => checklist.id === id);
  }
```

This function is a little different to the rest that we have added because this one actually returns something. This function will come in useful when we want to view the detail for a particular checklist. Earlier, we set up a route that would accept an `id` as a parameter. We can easily pass the `id` for a particular checklist through to the detail page, but we need *all* of the data for the checklist. This will allow us to easily get all of that data, whilst still only needing to pass the `id` through the URL. The `find` method we use here simply looks for a checklist with an `id` that matches the `id` passed into the `getChecklist`.

addItem

We have added plenty of functions for modifying the data of our checklists, now we will need some to modify the items within a checklist. As I mentioned before, you may prefer to separate this out into its own service.

> Modify the `addItem`, `removeItem`, and `renameItem` functions to reflect the following:

```
addItem(checklistId, data): void {

  this.getChecklist(checklistId).items.push({
    title: data.name,
    checked: false
  });

  this.save();
```

```
  }

  removeItem(checklist, item): void {

    let index = checklist.items.indexOf(item);

    if(index > -1){
      checklist.items.splice(index, 1);
      this.save()
    }

  }

  renameItem(item, data): void {

    item.title = data.name;
    this.save();

  }
```

This is all basically the same idea as modifying our checklists.

toggleItem

Each of the items in our checklist can be toggled between a completed and incomplete state. This function simply flips the value of a particular item and then saves it to storage.

> Modify the `toggleItem` function to reflect the following:

```
  toggleItem(item): void {
    item.checked = !item.checked;
    this.save();
  }
```

generateSlug

Finally, we come to perhaps out most interesting function. Let's take a look.

> Modify the `generateSlug` function to reflect the following:

```
  generateSlug(title): string {

    // NOTE: This is a simplistic slug generator and will not
handle things like special characters.
    let slug = title.toLowerCase().replace(/\s+/g, '-');

    // Check if the slug already exists
    let exists = this.checklists.filter((checklist) => {
      return checklist.id.substring(0, slug.length) === slug;
    });
```

```
    // If the title is already being used, add a number to make
the slug unique
    if(exists.length > 0){
      slug = slug + exists.length.toString();
    }

    return slug;

  }
```

As I mentioned, we need a unique `id` for each of our checklists, but we want the ids to look nice and make sense in the URL. To do this, we will be using a hyphenated version of the title for the checklist. However, if a user were to create two checklists with the same title then it would no longer be unique. This function checks if the `id` already exists, and if it does it will add a number to the end of the `id` slug. We check for any `id` that has a matching *start*, so if there are multiple checklists with the same `id` they will be numbered in a sequence like this:

- my-checklist1
- my-checklist2
- my-checklist3

and so on. We are using a very simple regex pattern to replace the spaces in the checklists title with hyphens, in a "serious" application you might want to modify this to be more robust.

We have finished implementing our checklist service - now, you will be able to import this service into any other page/component/service you like, inject it into the constructor, and then use it as you please!