



## Upgrading your project from Angular 5 to 6

Since recording the course, Angular 6 has been released as the latest version. This is a guide to upgrade your project to Angular 6 and successfully make it run. In order to check your Angular version you can run the following in the console, in your project's folder.

**ng -v.**

This should print something like this:

Angular CLI: 6.1.5

Node: 8.11.4

OS: win32 x64

Angular: 6.1.6

Notice that the version printed is Angular 6.1.6. You'll notice that trying to run **ng serve** will result in the following error:

*Local workspace file ('angular.json') could not be found*

That's because the new Angular version requires an **angular.json** file rather than **angular-cli.json**. We'll take care of this in a second.

### Step 1.

Update your Angular CLI globally and locally, and migrate the configuration to the new **angular.json** format by running the following:

- **npm install -g @angular/cli**
- **npm install @angular/cli**
- **ng update @angular/cli**

### Step 2.

Update all of your Angular framework packages to v6, and the correct version of RxJS and TypeScript:

- **ng update @angular/core**

### Step 3.

Update Angular Material to the latest version.

- **ng update @angular/material**



#### Step 4.

- **Use ng update or your normal package manager tools to identify and update other dependencies. This step should return no new packages in need to update for our project, but just as a safety net.**

#### Step 5 - Almost Done!

The upgrade itself is complete, we need to remove deprecated RxJS 6 features. In order to do so, run:

- **npm install -g rxjs-tslint**
- **rxjs-5-to-6-migrate -p src/tsconfig.app.json**

During the final step, you might encounter an error saying *Cannot find module 'typescript'*. In order to fix it, all you have to do is run the following and you're set!

- **npm install -g typescript**

When trying to run ng serve, you might encounter an error saying *Could not find module "@angular-devkit/build-angular"*. Same as before, try running the following, it should fix the issue.

- **npm install --save-dev @angular-devkit/build-angular**

#### Final Notes

The paths in the new angular.json files are a bit different. In case you get an error when trying to startup the server saying that it can't find the styles files, make sure that the paths in angular.json look like this:

```
"styles": [  
  "node_modules/bootstrap/dist/css/bootstrap.css",  
  "src/styles.css"  
]
```

That's all, congratulations!

For more resources about the upgrade, check out <https://update.angular.io/>



## Upgrading your project from Angular 6 to 8

Since recording the course, Angular 6 has been released as the latest version. This is a guide to upgrade your project to Angular 6 and successfully make it run. In order to check your Angular version you can run the following in the console, in your project's folder.

**ng -v.**

This should print something like this:

Angular CLI: 8.1.0

Node: 12.5.0

OS: win32 x64

Angular: 8.1.0

### !!Github notes!!

Sometimes when trying to run `ng update @angular/core` you will get an error saying that Repository is not clean. Please commit or stash any changes before updating.

In order to fix this, run:

- `Git add .`
- `Git commit -m "This is a test commit message"`
- Try rerunning `ng update @angular/core`
- Repeat this whenever you get the message during your upgrade

Notice that the version printed is Angular 8.1.0. You'll notice that trying to run **ng serve** will result in the following error:

*The serve command requires to be run in an Angular project, but a project definition could not be found.*

That's because the new Angular version requires an **angular.json** file rather than **angular-cli.json**. We'll take care of this in a second.

### Step 1.

Update your Angular CLI globally and locally, and migrate the configuration to the new **angular.json format** by running the following:

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



## Step 2.

Update all of your Angular framework packages to v6, and the correct version of RxJS and TypeScript:

- `ng update @angular/core`

## Step 3.

- **Use `ng update` or your normal package manager tools to identify and update other dependencies. This step should return no new packages in need to update for our project, but just as a safety net.**

## Step 4 - Almost Done!

When trying to run `ng serve`, you might encounter an error saying *Could not find module "@angular-devkit/build-angular"*. Same as before, try running the following, it should fix the issue.

- `npm install --save-dev @angular-devkit/build-angular`

## Final Notes

That's all, congratulations!

For more resources about the upgrade, check out <https://update.angular.io/>

**The instructions in the video for this particular lesson have been updated to use Angular 7. Please see the lesson notes below for the corrected instructions.**

Welcome to this course on **Angular** where we will learn the ins and outs by building our own **webapp**. In this lesson, we'll focus on setting up our environment.

## Getting Node.js and Angular

In order to use **Angular**, our first step is to **download** and **install node.js** (<https://nodejs.org/en/>). Make sure to download the **LTS** version, as this is the most stable.

**The following instruction has been updated, and differs from the video:**

For our project, the current **LTS** version will be version 10.



**Node.js** gives us **npm**, or **node package manager**, so now we're ready to get the **Angular CLI** (<https://angular.io/>). With **Angular CLI**, we can easily boot up our project without the need to create build scripts or any similar items to get our application working.

**Open** up **Terminal** or **Command Prompt** so that we can run our commands to install **Angular CLI**. We will first run the command below which will install the **Angular 7 CLI globally**.

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

**The following two instructions have been updated, and differ from the video:**

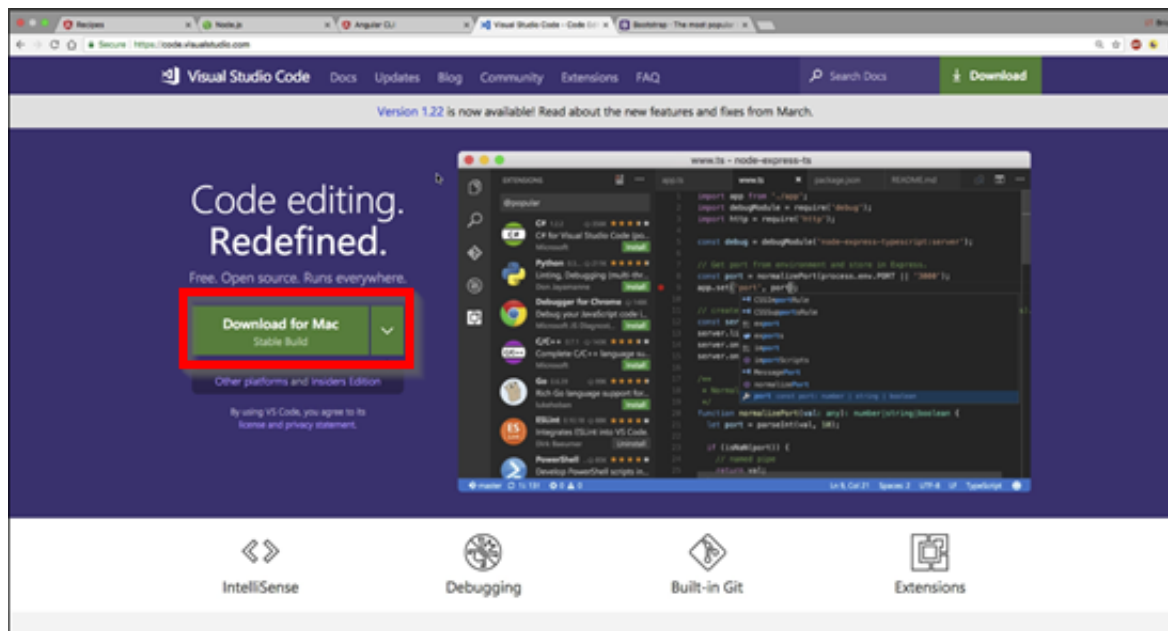
You may also wish to install **Angular 7 locally**:

```
npm install @angular/cli
```

If coming from an older version of **Angular**, you may instead choose to install the **1.7.4 CLI locally** for more compatibility:

```
npm install @angular/cli@1.7.4
```

While this is installing, head on over to **Visual Studio Code** (<https://code.visualstudio.com/>) and **download** and **install** it. This program is what will allow us to edit our code throughout the project.



## Creating our Project

Now we can create our workspace with our commands in **Terminal/Command Prompt**. Our first task is to create our **new** project, which we will call **recipe**.

```
ng new recipe
```

Once that completes installation, we need to move to the directory for our new project.

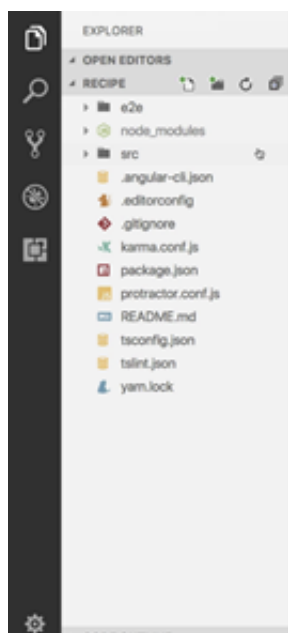
```
cd recipe
```

We can now open our project in **Visual Studio Code**. You can do this manually by starting the program and selecting **Open Folder**. Alternatively, you can simply use the following command to open it automatically:

```
code .
```



If all is well, you will see your open project!



Show All Commands ⇧ ⌘ P  
Go to File ⌘ P  
Find in Files ⇧ ⌘ F  
Start Debugging F5  
Toggle Terminal ^ `

In the next lesson, we'll talk a bit more about the files and folders automatically set up for us.

**If upgrading this project to Angular 7 from Angular 5 or 6, please check out the Upgrade Instructions for Angular 6 at the beginning of the course. The same steps will be involved.**

The explanations in the video for this particular lesson have been updated to use Angular 7. Please see the lesson notes below for the corrected explanations.

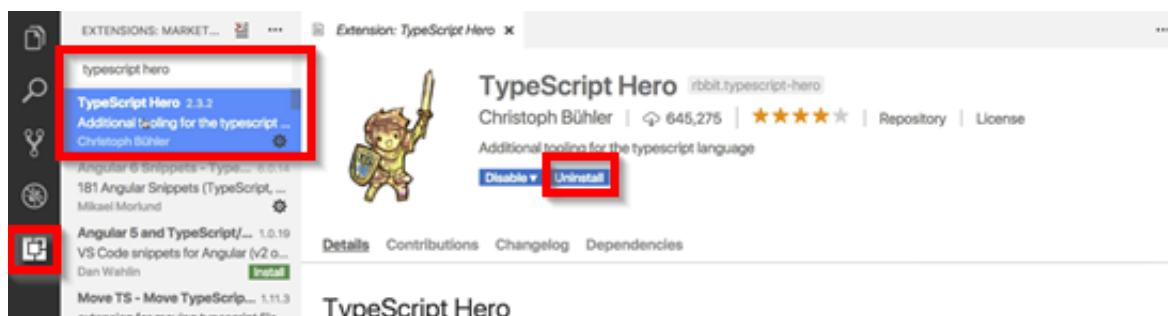
In this lesson, we're going to do a quick overview of the folder and file structure we can see for our project.

## Adding TypeScript Hero

Before we do anything, in **Visual Studio Code** you might first notice the left side bar. This side bar consists of five buttons:

- The Explorer
- The Search
- The Version Control Area
- The Debugging Area
- The Extensions

We want to install the **TypeScript Hero extension** before we start. Click the **last icon** on the side bar, **search** for **TypeScript Hero**, and **install** it if you don't already have it. Since **Angular** is **TypeScript** based, this will help us out with shortcuts and numerous other things that will make coding easier.



## About the Folders and Files

Back in the **Explorer**, we'll **right click** and **Hide** the **Open Editors** dropdown so that we can focus in on our actual project files. For our project, you will see three main folders. The first one, **e2e**, is used for testing, which is beyond the scope of this course. Meanwhile, **node\_modules** is for all our dependencies that came with **npm**. We'll skip **src** for a moment to talk about the files.

For most of our files, we will simply give a brief overview that tells you a little bit about what the file is for. Many of these are auto-generated or beyond this course's scope.

- **gitignore** is for version control and can instruct things like Git on what to ignore.
- **karma.conf.js** is used for testing, which is again beyond the scope of our course.
- **package.json** contains all our dependencies which are required to make the project work.
- **protractor.conf.js** is another file for testing.
- **README.md** contains information on running the program with command line information.
- **tsconfig.json** contains the compiler settings that tell the program how to **transpile** the **TypeScript** to **Javascript**, which is needed to be web compatible.
- **tslint.json** will check for errors or set restrictions, which you can personally specify if you so wish.
- **yarn.lock** or **package.lock** sets up the file dependencies and is auto-generated

The following explanation has been updated, and differs from the video:



However, one file to keep in mind is the **angular.json** file. In this file, we can **load** in our global external files, such as **Javascript files**, **CSS styles**, or **configuration files** as a small example.

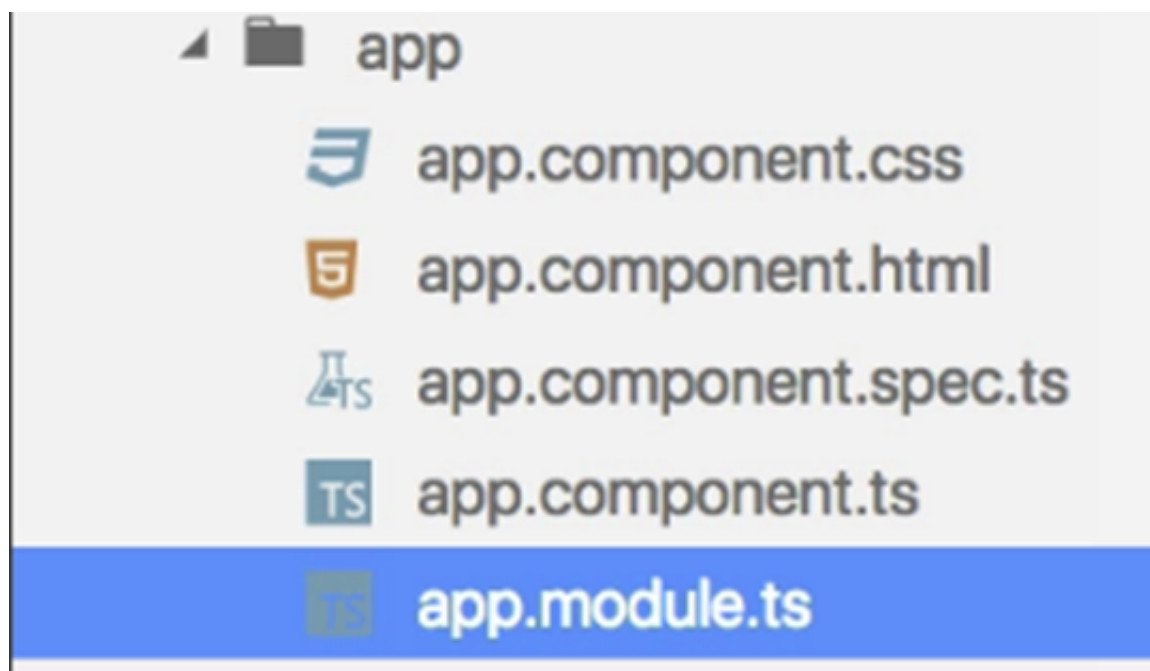
## The Src Folder

Let's talk about the **src folder** now. Several of the folders and files aren't relevant for right now, so we'll focus on the important ones.

- The **polyfills.ts file** contains information to help with browser compatibility.
- The **styles.css file** works exactly as you expect and allows us to style the **CSS**.
- The **favicon.ico** file can be replaced with our own custom **Favicon**.
- The **assets folder** will be used to contain all our images for our app.
- The **environments** folder contains files that let us alter the **configurations** or call **APIs**
- Last, the **index.html file** is the webpage on which our app will be built. You can see in the code below where it will be placed:

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

The most important folder in **src**, however, is the **app folder**. This is the core folder for building our **app**, and already includes some basic **component files** for us. Of these, **app.module.ts** is one we will use a lot to **import** our app's elements, declare them, and provide them. Overall, it is where our app gets structured to turn it into a workable package.



If you'd like to learn more about any of these files, please check out the **documentation** of the workspace: <https://angular.io/guide/file-structure>

In the next lesson, we'll begin adding in pictures and setting up some of our dependencies.



**The explanations in the video for this particular lesson have been updated to use Angular 7. Please see the lesson notes below for the corrected explanations.**

In this lesson, we're going to begin setting up some of our dependencies and installing our photos.

## Getting Bootstrap

So we can have an easy time with styling, we're going to install **Bootstrap** for our project. However, the standard **Bootstrap** package depends on **jQuery**, which we don't want to use in our app. However, by going to the following website, we can find a version of **Bootstrap** that was rewritten specifically to use **Angular**: <https://valor-software.com/ngx-bootstrap/#/>

Hit **Get Started** and you will be taken to a page where you can find the command line needed to install this version of **Bootstrap**.

# Installation instructions

Install **ngx-bootstrap** from **npm**

```
npm install ngx-bootstrap --save
```

**Copy** and **paste** the command line into **Terminal** or **Command Prompt**, though make sure you are in your project directory.

```
npm install ngx-bootstrap --save
```

**Optionally**, you can choose to include the **Bootstrap CDN** instead of installing **Bootstrap** in the following step. To do this, head over to the **src** folder and locate **index.html**. Then, **copy** and **paste** the **link** to the **Bootstrap CDN** under the **Favicon** reference in the **head tags**.

```
<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" rel="stylesheet">
```

**For our project**, we're just going to **install Bootstrap directly** by using the following command:

```
npm install --save bootstrap
```

We will also install some helper functions by using the following command to get **lodash**.



```
npm install --save lodash
```

## Setting up Dependencies

**The following explanation and code has been updated, and differs from the video:**

Now that we've installed these necessary elements, we need to add in the dependency calls for them. Locate **angular.json** and **open** it in **Visual Studio Code**. Find a reference to **styles**, where you should see **styles.css** already referenced. In this block, we're going to add a reference to our new **Bootstrap file** which is located in the **node\_modules** folder. Remember to hit **save** when you're finished.

```
"styles": [  
  "node_modules/bootstrap/dist/css/bootstrap.css",  
  "src/styles.css"  
],
```

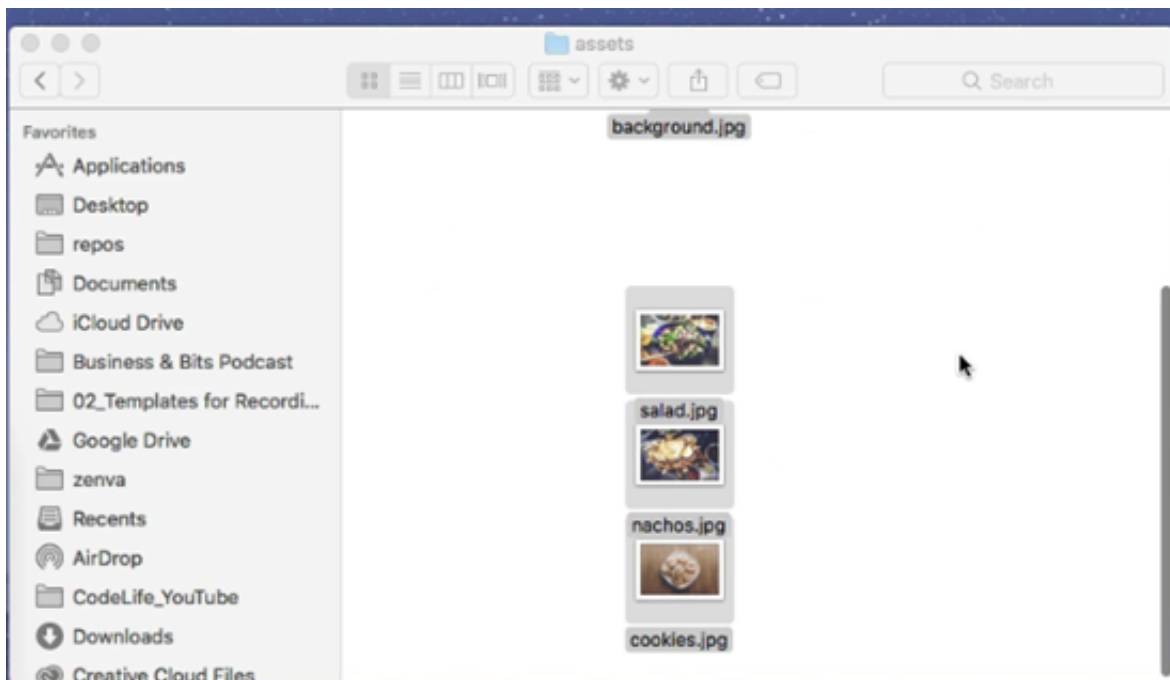
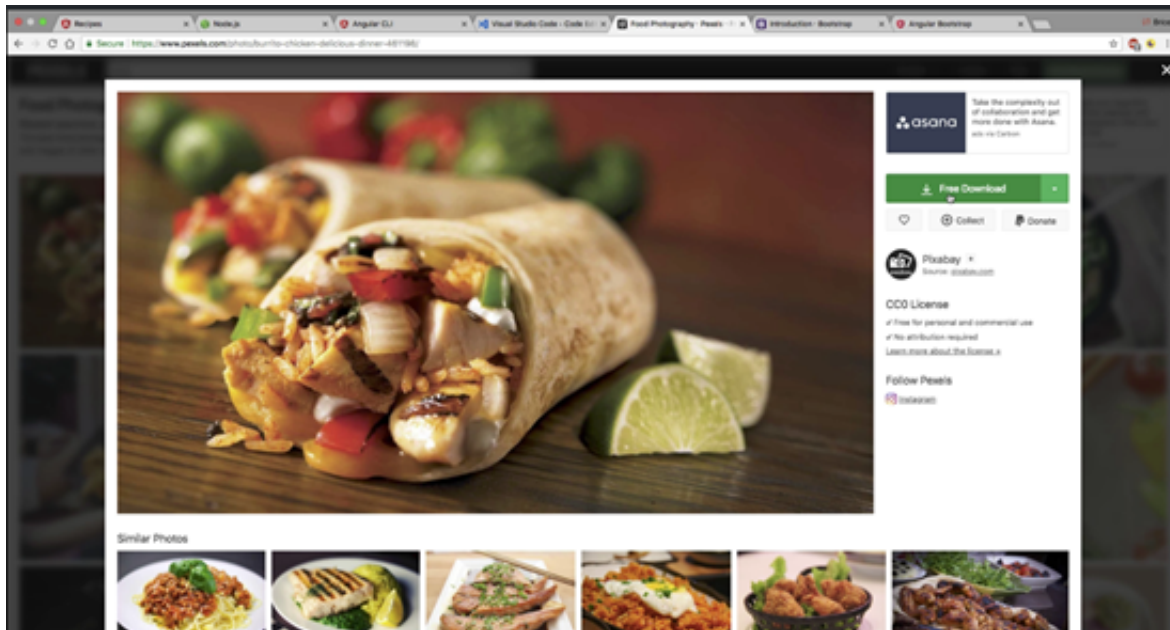
(**Note:** You can hit Command + P on Mac or Control + P on Windows to open up a search bar in the program's Explorer to find the file).

Next, we're going to head on over to **typings.d.ts**. We want to be able to reference any external **JSON** files later in the project, so we're going to create a **TypeScript declaration** for it. With the setup below, the program will know how to open any type of **JSON** and be able to export it for use elsewhere.

```
declare module "*.json" {  
  const value: any;  
  export default value;  
}
```

## Adding Photos

The last thing we want to do is add some photos to our **assets folder** in the **src folder**. If you would like to use the pictures we're using, you can **download** the **Course Files** and copy them from the **assets folder** in there. If you would like to select your own, head on over to **Pexels** (<https://www.pexels.com/>) and search for "food." The photos on **Pexels** are mostly free for personal and commercial use, so you can choose what you would like. Likewise, make sure to put them into your **assets folder** for the project.

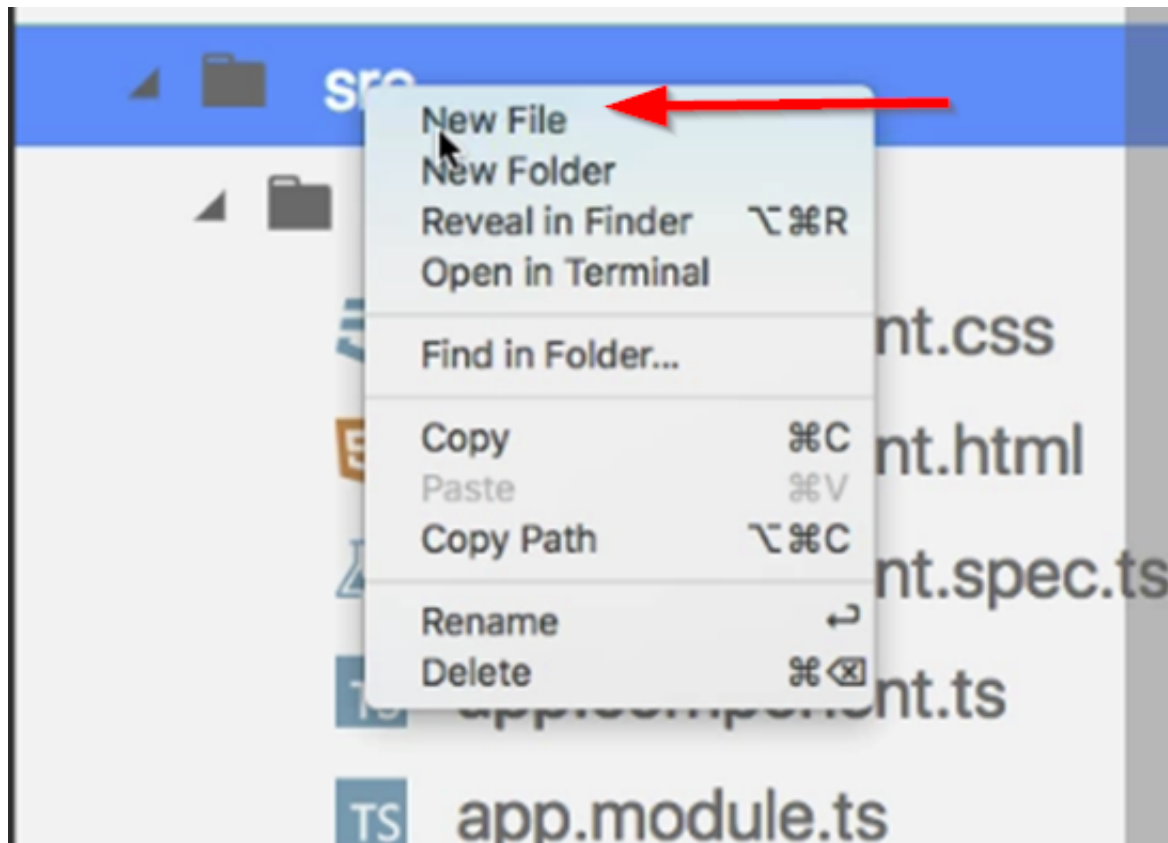


In the next lesson, we can finally start adding some data.

In this lesson, we're going to create the data we're going to work with in our project.

## Creating our Recipes with JSON

The best way to create and store our data for this project is through a **JSON** file. In our **src** folder, we're going to right click and create a **New File** called **data.json**.



In our new **data.json** file, we're going to start writing out our data. Since our app is going to be based around recipes, it makes sense our data will consist of an **array** for **recipes**. In this array, we'll create three objects to contain each of our recipes. Each object will contain several properties with declared values. There will be an id for the recipe, the recipe's title, the recipe's description, the recipe's serving amount, and the recipe's image URL. Further, we will also have two more **arrays in each recipe object**. One **array** will contain the instructions as **strings**, while another will contain the ingredients. In our ingredients **array**, we'll use objects so we can hold both the amount of the ingredient we need as well as the name of the ingredient.

```
{
  "recipes": [
    {
      "id": 1,
      "title": "Best Salad in the World",
      "description": "This is a recipe for the best tasting salad in the world. It has lots of carrots and lettuce!",
      "serves": "2 people",
      "imageUrl": "assets/salad.jpg",
      "ingredients": [
        {
          "amount": "1 head",
          "name": "lettuce"
        }
      ]
    }
  ]
}
```



```
    },
    {
      "amount": "10",
      "name": "carrots"
    },
    {
      "amount": "2 tablespoons",
      "name": "ranch dressing"
    }
  ],
  "instruction": [
    "remove lettuce leaves",
    "add lettuce leaves to a large bowl",
    "then add the carrots the bowl",
    "add the ranch dressing",
    "then mix ingredients together in the bowl"
  ]
},
{
  "id": 2,
  "title": "Chocolate Chip Cookies",
  "description": "This was a recipe handed down to me by my grandmother Ruth who use to make the best chocolate chip cookies!",
  "serves": "4 people",
  "imageUrl": "assets/cookies.jpg",
  "ingredients": [
    {
      "amount": "1 bag",
      "name": "chocolate chips"
    },
    {
      "amount": "2",
      "name": "eggs"
    },
    {
      "amount": "1 cup",
      "name": "flour"
    },
    {
      "amount": "1 cup",
      "name": "sugar"
    }
  ],
  "instruction": [
    "get a large bowl",
    "crack eggs into the bowl",
    "add the rest of the ingredients to the bowl",
    "mix ingredients with a whisk",
    "then evenly place small clumps of cookie dough on a baking sheet",
    "place baking sheet in oven for 12 minutes at 350 degrees F"
  ]
},
{
  "id": 3,
  "title": "Cinco de Nachos",
```



```
"description": "I discovered this recipe when I went on a trip to Mexico. A nice lady taught me how to make world famous nachos.",
"serves": "2 adults or 4 kids",
"imageUrl": "assets/nachos.jpg",
"ingredients": [
  {
    "amount": "1 bag",
    "name": "tortilla chips"
  },
  {
    "amount": "1 pound",
    "name": "shredded cheese"
  },
  {
    "amount": "1 can",
    "name": "refried beans"
  }
],
"instruction": [
  "spread tortilla chips on a large plate",
  "warm up beans in a pot on the stove",
  "sprinkle the shredded chesse on top of the chips",
  "place plate in the oven for 10 minutes at 350 degrees F",
  "finally remove plate from oven and spread refriend beans on top of the chips"
]
}
```

**(Note:** Make sure to use double quotes and not single quotes, as single quotes are not valid for **JSON**).

Overall, we'll wind up with the code above and have a large chunk of data that is structured in a way we can work with for our app. You may feel free to set your own values and create whatever recipes you want. However, please make sure the **imageUrl** matches the name of your image and that you stick to our data structure for the sake of following along. Also, don't forget to **save** the file.

In the next lesson, we'll start working with the **classes** and **interfaces** we'll be using with our app.

In this lesson, we're going to create a **class** and **interfaces** for our data.

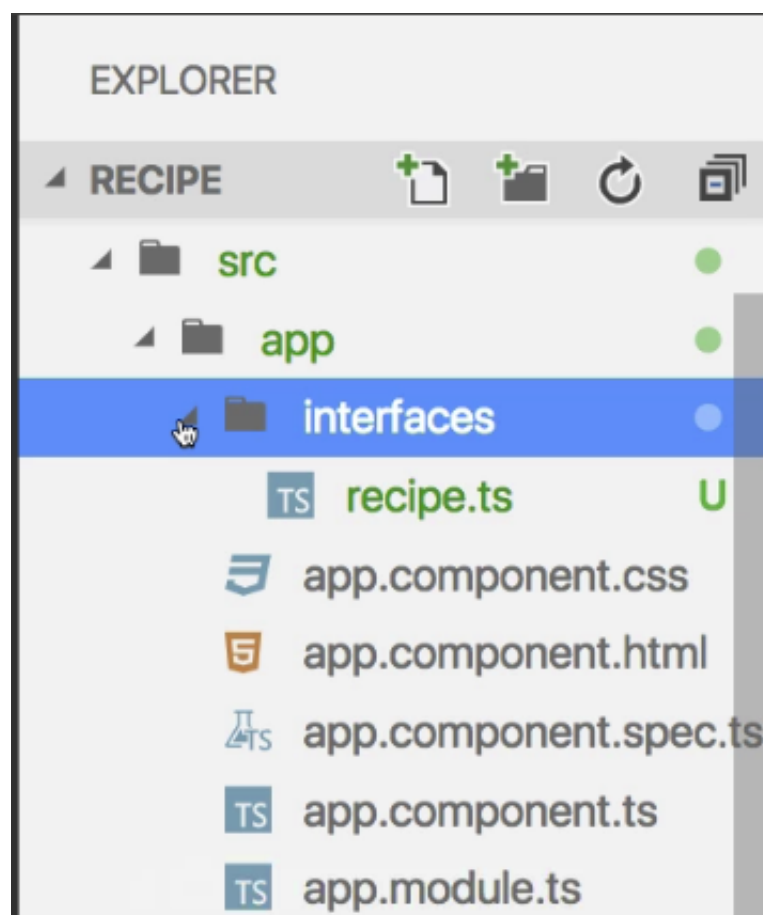
## Defining our Interfaces

For demonstration purposes, we're going to use a command in **Terminal/Command Prompt** to generate our interface. With the following command, we can generate (g) an **interface** in a folder called **interfaces** with the title of **recipe**.

```
ng g interface interfaces/recipe
```

You can see that the folder and file is generated for you automatically to get a start from.

```
~/zenva/code/recipe > master ng g interface interfaces/recipe  
create src/app/interfaces/recipe.ts (28 bytes)  
~/zenva/code/recipe > master
```



Since our ingredients from our data are also objects, we'll create an **interface** for them as well.

```
ng g interface interfaces/ingredient
```

We'll work on **ingredient.ts** first, where you'll see some code already provided. So that our



**class** and **interfaces** won't conflict name wise, we will make sure to rename our **interface** to **IIngredient**. The use of "i" is a standard convention many programmers use to differentiate their **interfaces** and **classes**. Now we can use our **interface** to define the **data types** of our "amount" and "name" properties that we used in our ingredient objects.

```
export interface IIngredient {  
  amount: string;  
  name: string;  
}
```

Similarly, we'll also do the same in **recipe.ts**. Once again, we'll change the name to **IRecipe** and add **data types** to all our properties. However, since we have a separate **interface** for **ingredients**, we need to import that. This allows us to set **IIngredient[]** as the **data type** for **ingredients**, which in turn uses our **IIngredient interface** to determine the **data types** for those elements.

```
import { IIngredient } from '../ingredient';  
  
export interface IRecipe {  
  id: number;  
  title: string;  
  description: string;  
  serves: string;  
  imageUrl: string;  
  ingredients: IIngredient[];  
  instructions: string[];  
}
```

## Defining our Recipe Class

Now that we have our **interfaces**, we can move on to setting up our **recipe class**. Once again, we'll use a command in **Terminal/Command Prompt** to generate the folder and file for us.

```
ng g class classes/recipe
```

In our new **recipe.ts** file for our **recipe class**, we're going to first **import** both our interfaces. This way we can utilize them for the **class**. We also need to command our **class** to **implement** the **interface**.

```
import { IRecipe } from '../interfaces/recipe';  
import { IIngredient } from '../../interfaces/ingredient';  
  
export class Recipe implements IRecipe{  
}
```

Since we've essentially made a contract with our **interface** to have the same properties, we need to once again declare the **properties** and **data types** in our **class** as well. These properties will be



**public**, which is inferred when we type nothing.

```
import { IRecipe } from '../interfaces/recipe';
import { IIngredient } from '../interfaces/ingredient';

export class Recipe implements IRecipe{
  id: number;
  title: string;
  description: string;
  serves: string;
  imageUrl: string;
  ingredients: IIngredient[];
  instructions: string[];
}
```

Last but not least, we'll create a **constructor** for our **recipe class**. However, opposite to the name, we're going to **destructure** the **data** that gets passed in. In other words, we're going to pass in an object with all the properties we have and register the class properties as being equal to them. This will make more sense later on as we start building our app more.

```
constructor({id, title, description, serves, imageUrl, ingredients, instructions}) {
  this.id = id;
  this.title = title;
  this.description = description;
  this.serves = serves;
  this.imageUrl = imageUrl;
  this.ingredients = ingredients;
  this.instructions = instructions;
}
```

We now have our **class** and **interfaces**. If you'd like to learn more about classes or interfaces in **TypeScript**, you can check the following documentation:

- Interfaces: <https://www.typescriptlang.org/docs/handbook/interfaces.html>
- Classes: <https://www.typescriptlang.org/docs/handbook/classes.html>

In the next lesson, we'll start writing the **services** for our app.



In this lesson, we're going to work on creating the **services** for our app.

## About Services in Angular

In **Angular**, **services** are used to pass data around. This can consist of passing information from **APIs** to the app or even passing data around between **components** in the app. For our app, the data we'll be passing around is our **JSON** file of recipes. With **services**, a lot of the **CRUD** work is handled by them instead of the **components**.

## Creating the Recipe Service

In **Terminal/Command Prompt**, we're going to use our generation command again to create our **service** for **recipe**.

```
ng g service services/recipe
```

This time you'll see two files have been generated. The **recipe.service.spec.ts** file is for testing, which you can ignore for this course. **Recipe.service.ts** is the file we will concern ourselves with later.

Back in **Visual Studio Code**, we need to make our **service** available to all our **components**. To do this, we need to make it a **provider**. Locate the file **app.module.ts** in the **src folder**. Near the top, above the **import** for **App Component**, we'll add a line to **import** our **recipe service**.

```
// services
import { RecipeService } from '../services/recipe.service';

// components
import { AppComponent } from '../app.component';
```

Next, we'll turn it into a **provider** by locating the appropriate **provider field** in **@NgModule**.

```
providers: [
  RecipeService
],
```

To make sure things are working, we'll open our app in our browser. We can do this by typing in the following command to **Terminal/Command Prompt**:

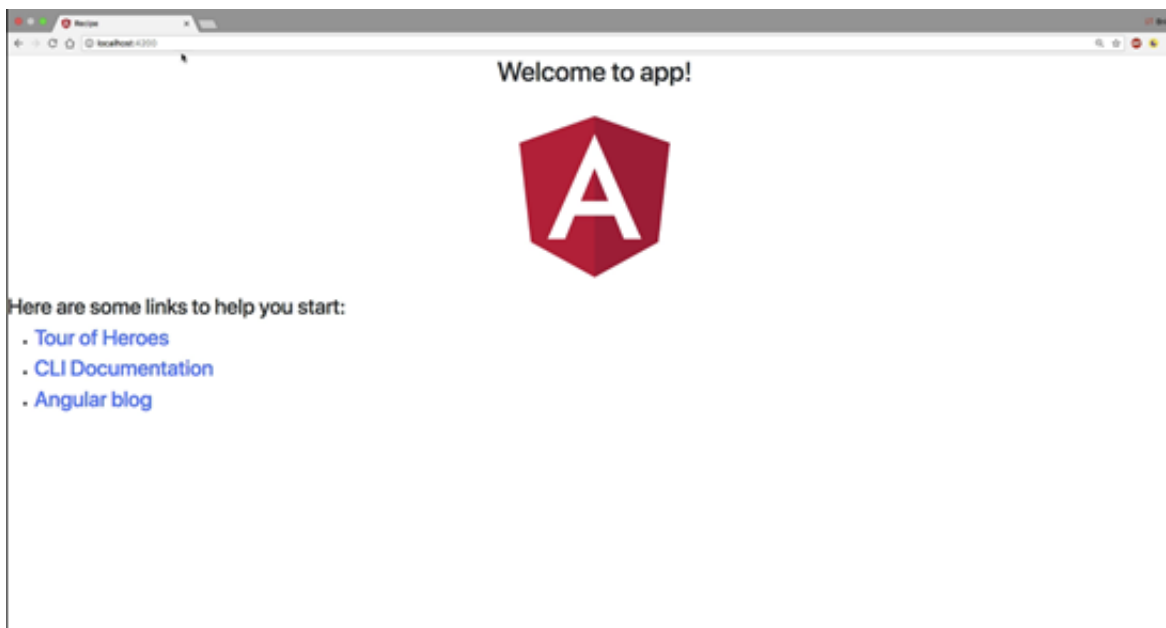
```
ng serve
```

After, open up your **browser** and go to **localhost:4200**. If all is well, you'll see your app built.



```
~/zenva/code/recipe master ng serve
** NG Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
Date:
Hash:
Time:    ms
chunk {inline} inline.bundle.js (inline) 3.85 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 20.4 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 554 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 425 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.43 MB [initial] [rendered]

webpack: Compiled successfully.
```



## Working on the Service

Now that we know our app can compile, let's head into **recipe.services.ts** and start creating our actual **service**. At the top of this file, the first thing we want to do is **import** our **recipe class**, our **ingredient interface**, our **data** from our **JSON file**, and **lodash**. Make sure to leave the **import** for **Injectable** as is.

```
import { Recipe } from '../classes/recipe';
import { IIngredient } from '../interfaces/ingredient';
import { Injectable } from '@angular/core';
import * as RecipeData from '../data.json';
import * as _ from 'lodash';
```

In the **@Interjectable** area above the **constructor**, we'll create a **private array** of recipes that will be used to store our recipes.

```
private recipes: Recipe[] = [];
```

Then, with our **constructor**, we'll do the actual adding to our **array**. To do this, we create a **forEach loop** which will pull the recipes from our **JSON** data file and **push** the recipes into our array we created above. The use of **<any>** is needed in this case to declare a data type, though it is a



specific case usually for **JSON** files.

```
constructor() {  
  (<any>RecipeData).recipes.forEach( recipe => {  
    this.recipes.push( new Recipe(recipe));  
  });  
}
```

Now that we have a container for all our recipes, we can create functions for our **components** to use. Outside of the **export class RecipeService** but **inside @Injectable**, we'll create our first function which will be able to retrieve all the recipes. For this, all we need to do is **return** the array we made earlier.

```
public getRecipes(): Recipe[] {  
  return this.recipes;  
}
```

In the next lesson, we'll continue creating our service so that we can have more of our **CRUD** endpoints.

In this lesson, we're going to finish up our **recipe service**.

## Getting One Recipe

Last lesson, we created our first function that is able to grab *all* our recipes. However, what if we want one recipe? For this, we can create a **function** that is able to get our recipes by their ID. We can use **lodash's find function** to iterate through all the recipe(s) in our **private recipe array** to find which one has the matching ID we pass in. Then, all we have to do is return the matching recipe.

```
public getRecipeById(id: number): Recipe {  
    return _.find(this.recipes, (recipe) => recipe.id === id);  
}
```

## Adding a Recipe

For our next function, we want to be able to create a recipe. Before we create the **function** for that, we need to create a separate **function** for our IDs. We want our **IDs** to auto-generate, which we can do with the code below. In this code, we use **lodash** to iterate over all our recipe IDs and determine which number is the maximum. Then, from a **saved variable** for that data, we add 1 to the ID we received back and return the value.

```
private getNextId(): number {  
    const max = _.maxBy( this.recipes, (recipe) => recipe.id);  
    return max.id + 1;  
}
```

Now we can create the function for creating a recipe. Disregarding ID, this means we need to take in all our properties that we used in our **JSON data** file as parameters. Of course, since we imported our **ingredient interface**, we can specify that to declare the data types for **ingredients**.

```
public createRecipe(  
    title: string,  
    description: string,  
    serves: string,  
    imageUrl: string,  
    ingredients: IIngredient[],  
    instructions: string[]  
) {  
  
}
```

For the actual functionality of the **function**, we want to store all these parameters into a single **object**, while also including our auto-generated ID. Then, we cast our object as a **Recipe type object** for our **array** and push it to our **recipes array**.

```
public createRecipe(  
    title: string,  
    description: string,
```



```
serves: string,
imageUrl: string,
ingredients: IIngredient[],
instructions: string[]
) {
  const newRecipeData = {
    id: this.getNextId(),
    title,
    description,
    serves,
    imageUrl,
    ingredients: [...ingredients],
    instructions: [...instructions]
  }

  const newRecipe = new Recipe(newRecipeData);

  this.recipes.push(newRecipe);
  return newRecipe;
}
```

## Updating and Deleting a Recipe

The last two **functions** we want to create are a function to **update** a recipe and a function to **delete** a recipe. For both of these, we will take in a **recipe** and **find the index** for the recipe passed in by matching the IDs. For our **update function**, we can then return the recipe at that index. Meanwhile, to **delete** our recipe, we need to first check if the **function** found anything (**lodash** returns -1 if there is no matching item). Once that's confirmed, we can **remove/splice** a single recipe at that index.

```
public updateRecipe(recipe: Recipe): Recipe {
  const recipeIndex = _.findIndex(this.recipes, (r) => r.id === recipe.id);

  this.recipes[recipeIndex] = recipe;
  return recipe;
}

public deleteRecipe(id: number): void {
  const recipeIndex = _.findIndex(this.recipes, (r) => r.id === id);

  if (recipeIndex !== -1) {
    this.recipes.splice(recipeIndex, 1);
  }
}
```

Our services are now complete and we can start working on some of our **components**. Remember to **save** your progress!

If you'd like more information about **lodash functions**, please feel free check out the **lodash documentation**: <https://lodash.com/docs/4.17.11>



Additionally, you can also check out **TypeScript's documentation** on creating functions if any item was confusing: <https://www.typescriptlang.org/docs/handbook/functions.html>





In this lesson, we're going to learn about generating and routing to pages in our app.

## Generating our Components

Head over to **Terminal/Command Prompt**, and make sure you're in your project folder. We're going to generate some **components** for our **app**. Next lesson will focus on creating the **navbar**, so we'll create the **component** for that first.

```
ng g component components/navbar
```

Next, we want to generate several pages for our webapp, including a home page, a page for editing recipes, a page for showing recipes, and a page for adding new recipes. For the sake of organization, we're going to give them their own **pages folder** too. To do this, run each of the four lines below separately.

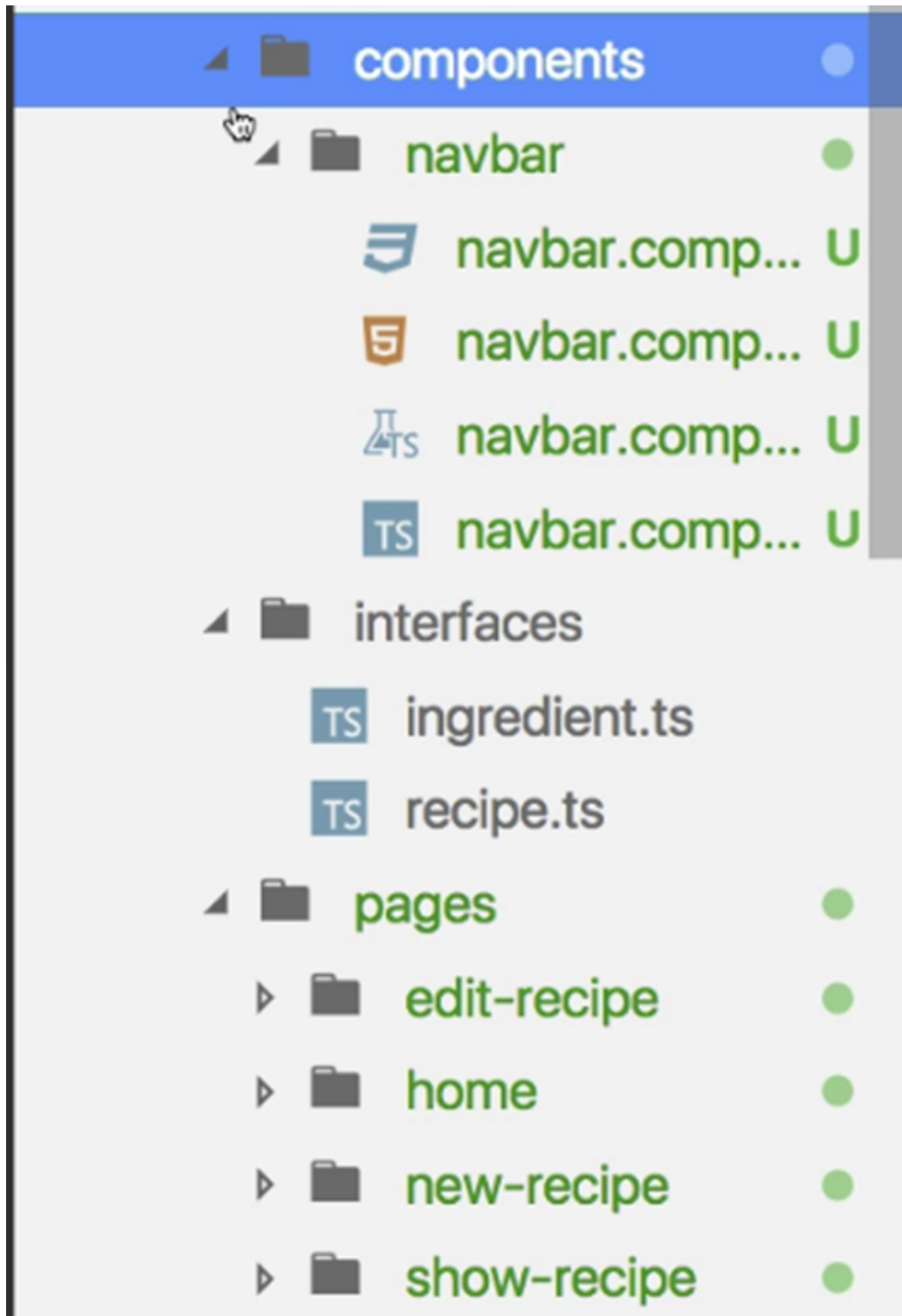
```
ng g component pages/home
```

```
ng g component pages/edit-recipe
```

```
ng g component pages/new-recipe
```

```
ng g component pages/show-recipe
```

Back in **Visual Studio Code**, you can see all our generated materials.



## Routing our Pages

Let's talk about routing. Within our **app folder**, we need to create a **New File** called



**app-routing.module.ts.** In this file, we're going to set up the **routing** so we can be *routed* to the correct pages by a designated path. In the following code, we're going to **import** our **NgModule**, **Routes**, and **RouterModule** so we can utilize their functions. We'll also **import** our **components** so that this file can see them. With an **@NgModule** declaration, we'll also set up our basic routing logic. We need to define our routes to be for our **root application** in this case since our pages are for our main application. In larger applications you might find other specifications because of the modular nature of this method of programming.

```
import { NewRecipeComponent } from '../pages/new-recipe/new-recipe.component';
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from '../pages/home/home.component';
import { EditRecipeComponent } from '../pages/edit-recipe/edit-recipe.component';
import { ShowRecipeComponent } from '../pages/show-recipe/show-recipe.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModuleModule { }
```

Now that we have these basic functions set up, we need to define the “**routes**” that are referenced for our **root**. To do this, all we need to do is store the **paths** and matching **components** in an **array of routes** above our **@NgModule**. For our paths, we use simple nomenclature that will instruct the app which component to go to given a certain web path. For **edit** and **show**, we also use the id parameter since we'll be matching IDs to create those pages. Last, we need to feature a few catch-alls to send us to the home page if all else fails.

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: 'recipes', component: HomeComponent},
  {path: 'recipes/new', component: NewRecipeComponent},
  {path: 'recipes/edit/:id', component: EditRecipeComponent},
  {path: 'recipes/:id', component: ShowRecipeComponent},
  {path: '**', component: HomeComponent}
];
```

Go to **app.module.ts** now, and you'll notice that all our **components** have been auto **imported** and made **declarations**. However, above our area for **services**, we need to manually **import** our **AppRoutingModule** that we exported.

```
import { AppRoutingModuleModule } from '../app-routing.module';
```

In the **@NgModule** section in this file, we also need to add it into the **imports**.

```
imports: [
  BrowserModule,
  AppRoutingModuleModule,
```

```
],
```

## Testing the Routing

One last thing we need to do is locate the **app.component.html** file in the **app** folder. You will notice all the material we see right now when testing our app. **Delete everything in this file** so we have a clean slate. For this file, all we need to do is declare a space for our **router outlet**, since we'll use our **components** for building the pages.

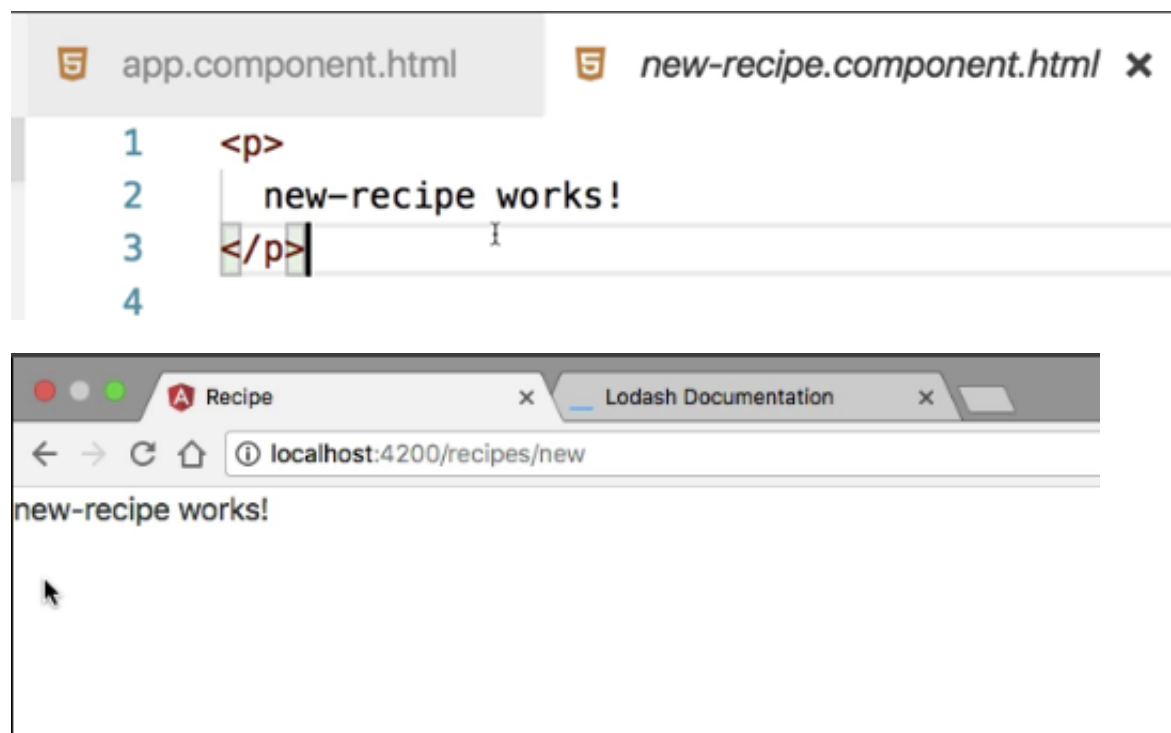
```
<router-outlet></router-outlet>
```

(**Note:** Though beyond the scope of this lesson, the above code is useful if you had children to your routes. This allows you to route and nest different components into different parts of your page based on the path).

Before testing, go into **app.component.ts** and **delete** the following line too:

```
title = 'app';
```

If you use **ng serve** to test your app, you can see that our pages can route correctly!



In the next lesson, we'll work on the navbar for our app.

In this lesson, we're going to work on our **navbar component**. For this section, you may find it helpful to get **Emmet** for auto completion (<https://emmet.io/>).

## Creating the Navbar

To begin, go to the **navbar folder**, open the **navbar.component.html** file, and **delete** the current page contents. Creating our page is as simple as writing regular HTML for a **navbar**. However, since we're using **Bootstrap** for this app, we will use **Bootstrap classes** to automatically style and prepare the bar. We'll begin with our starting tags which will make the navbar and color it for us. We'll also add in the brand/name for our page.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
  <a href="" class="navbar-brand">Tacos & More</a>
</nav>
```

Under our brand, we'll create a **div** with classes for **collapse responsiveness**, followed by an **unordered list** with some special **margin classes**. We only want one link/list item for this navbar, in this case to add a new recipe. We'll add that now as well.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
  <a href="" class="navbar-brand">Tacos & More</a>
  <div class="collapse navbar-collapse">
    <ul class="navbar-nav ml-auto mt-2">
      <li class="nav-item active">
        <a href="" class="nav-link">Add Recipe</a>
      </li>
    </ul>
  </div>
</nav>
```

We have most of our code, but we need to do something special with our links. Rather than have the browser refresh, we want our links to use our router to go to their correct pages. As such, we need to change the **href** property to be **routerLink** and use the **paths** we defined to tell the link which page to route to.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
  <a routerLink="" class="navbar-brand">Tacos & More</a>
  <div class="collapse navbar-collapse">
    <ul class="navbar-nav ml-auto mt-2">
      <li class="nav-item active">
        <a routerLink="/recipes/new" class="nav-link">Add Recipe</a>
      </li>
    </ul>
  </div>
</nav>
```

**(Note:** If you'd like to learn more about any of the Bootstrap classes we've used so far, please check out the Bootstrap documentation: <https://getbootstrap.com/docs/4.3/components/navbar/>)

## Adding the Navbar to Home

Our navbar is done, but we need to add it into our home page. In the **pages folder**, locate the **home.component.html** file, open it, and **delete** the page contents like before. Now we can add our **navbar**. With **Angular**, each **component** we create comes with a **selector**, which can be seen in the **.ts file** for each **component**. Though you can name it yourself, **Angular** will auto-generate a name for us to use as well. With this **selector**, we can quickly and easily embed the **component** into a **different component**.



Our selector for the **navbar** is **app-navbar**, so we simply need to add it to our **home.component.html** file like so:

```
<app-navbar></app-navbar>
Home Works
```

Testing the app, you'll see the page works as expected!



Before we close out, make sure to **add the navbar to all the other components in pages** in the same way. This way we can see the navbar no matter which page we're on.

In the next lesson, we'll start building up our home page!

The instructions in the video for this particular lesson have been updated to use Angular 7. Please see the lesson notes below for the corrected instructions.

In this lesson, we're going to begin making our home page.

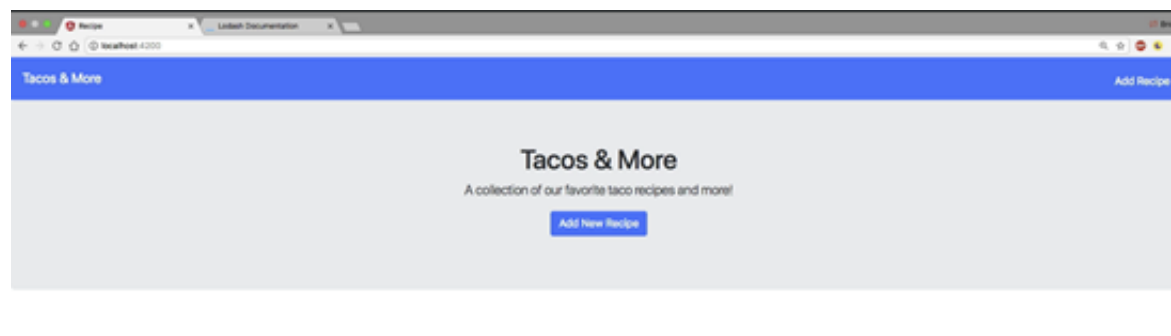
## Adding a Jumbotron

Over in **home.component.html**, we'll make sure our page is clean **except** for the navbar. The first element we're going to add in is a **jumbotron**. Once again, we'll be using many Bootstrap classes, which you can find out more about here: <https://getbootstrap.com/docs/4.3/components/jumbotron/>

With the code below, we're going to contain our **jumbotron** so it doesn't hit the sides and center the whole object. Our **jumbotron** will have a **heading**, as well as some paragraphs. For the first one we'll add some copy, as well as use **flexbox** logic to **center** and display the text responsively. Our second paragraph will have a link to add a new recipe. Like our navbar, we need to use a **routerLink** in order to route our page correctly.

```
<app-navbar></app-navbar>
<section class="jumbotron text-center">
  <div class="container">
    <h1 class="jumbotron-heading">Tacos & More</h1>
    <div class="d-flex justify-content-center">
      <p class="lead w-50">
        A collection of our favorite taco recipes and more!
      </p>
    </div>
    <p>
      <a routerLink="/recipes/new" class="btn btn-primary">Add New Recipe</a>
    </p>
  </div>
</section>
```

Using **ng serve** and viewing the page again, you should see we have our **jumbotron** mostly in place.



## Changing CSS to SCSS

The following instructions for this section have been updated, and differ significantly from the video:

Unfortunately, when we built this application, we set up **CSS** files instead of **SCSS** files. We need to fix this before we can style our **jumbotron**. We'll first change the project's default format by using



the following command in **Terminal/Command Prompt**. Again, be sure you're in your project directory before running the command. Keep in mind this command only works for **Angular 6** and **Angular 7**:

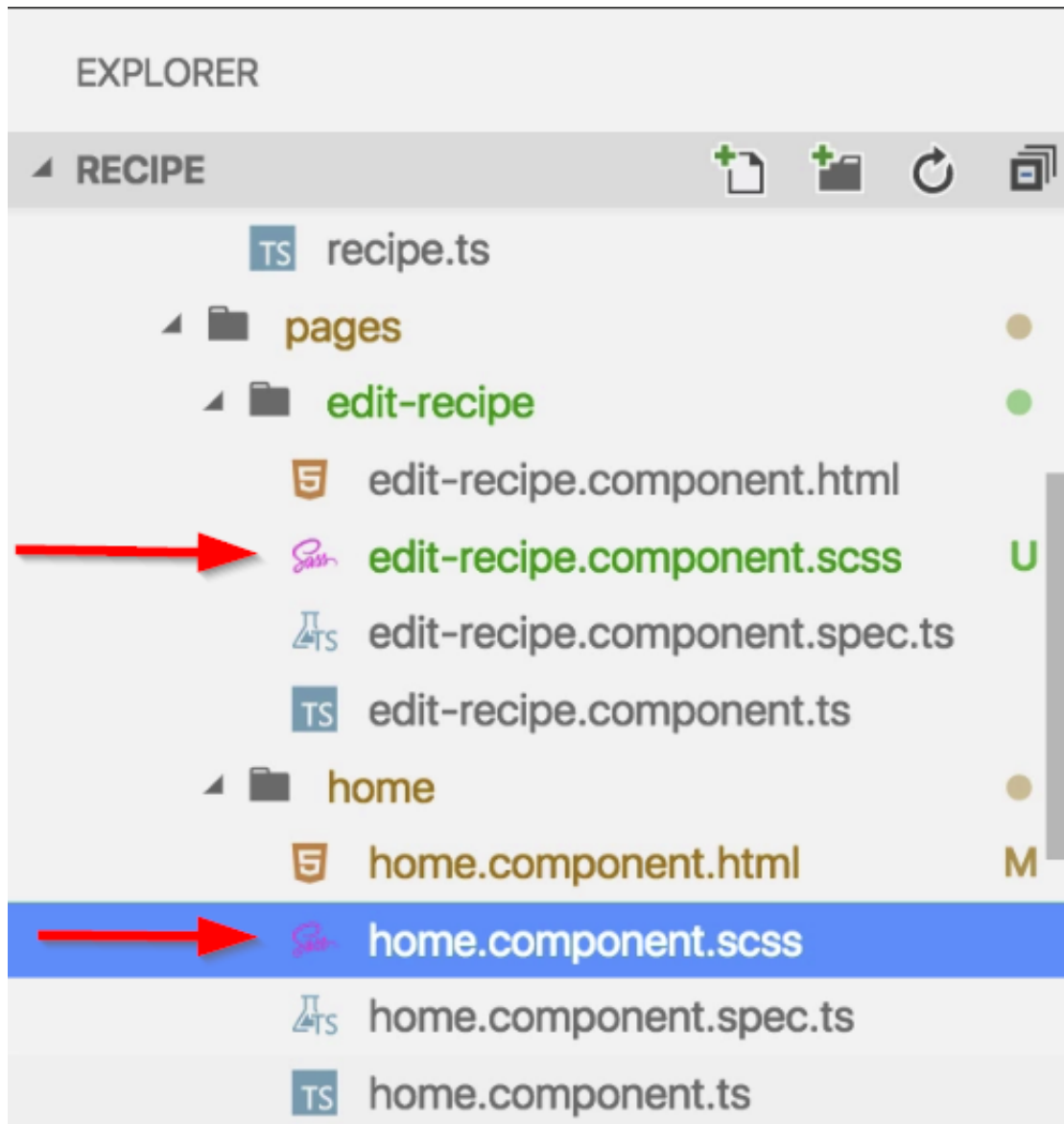
```
ng config schematics.@schematics/angular:component.styleext scss
```

Open up **angular.json** next and find the **styles** in the **projects** area near the top. We need to change it to the following:

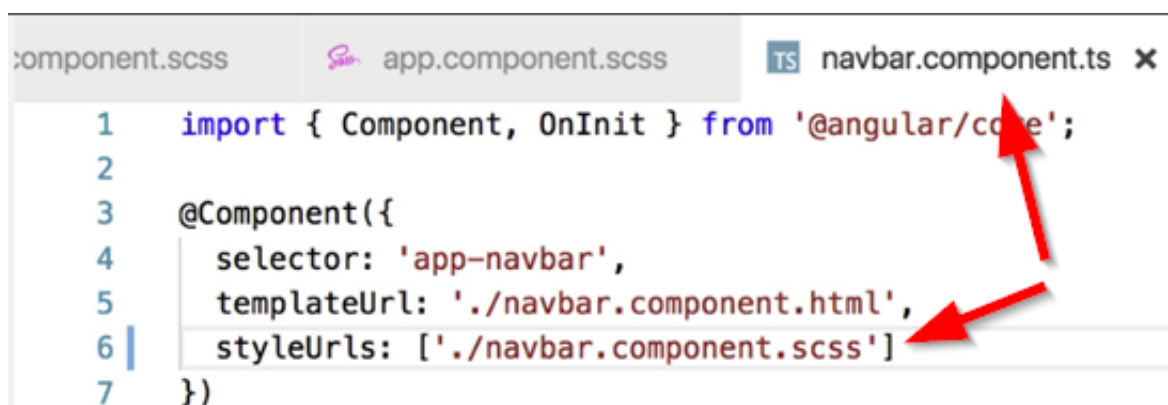
```
"styles": [  
  "node_modules/bootstrap/dist/css/bootstrap.css",  
  "src/styles.scss"  
],
```

With all that changed, we now have to go through each of our **components manually** and **change** the file extension to be **.scss**. Make sure to do this for the navbar, home page, edit recipe page, new recipe page, show recipe page, and the app.component page.





Since we changed the extension, we also have to go to **each components .ts file** and **change** the **styleURLs** extension there as well so they import correctly. You can make this easier by **opening the search from the left side bar** and **searching for .css**. Make sure to **not change** the bootstrap extension in **angular.json** or the extension in **app.po.ts** though!

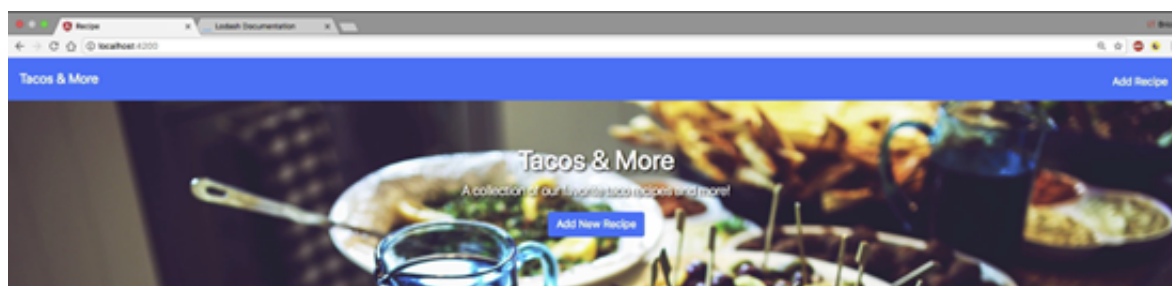


## Styling the Jumbotron

With this taken care of, we can now actually style our **jumbotron**. Locate **home.component.scss** and open it up. Fortunately, every style sheet is scoped to the particular component it's for, so we don't need to do any special wrapping for our styling. For our **jumbotron**, we want to add one of the images from our **assets** and change our text a bit, particularly making it white so it's more visible.

```
.jumbotron {  
  background: url('/assets/food-background.jpg');  
  background-size: cover;  
  border-radius: 0;  
}  
  
.jumbotron-heading, p.lead {  
  color: white;  
  text-shadow: 2px 2px 5px #000000;  
}
```

Our **jumbotron** looks less plain now~!



In the next lesson, we'll finish up our home page by adding cards for all our recipes.

In this lesson, we're going to finish up the home page for our web app!

## Component Importing

First, we need to bring in our **recipe service**, as this service contains all the data we need to display our recipes on the home page. In **home.component.ts**, we need to locate the **HomeComponent class** and alter the **constructor** to use the **recipe service** as follows:

```
constructor(public recipeService: RecipeService) { }
```

(**Note:** This one does need to be directly specified as public to work and can't be inferred).

At the top of the page, you should now see an **import** statement *automatically generated* for **RecipeService**, so we don't need to do anything else.

## Creating Cards

Back in **home.component.html**, we will add a new section below our **jumbotron**. We'll start this section off with a simple **container** and **row** from **Bootstrap** to contain our section.

```
<section class="container">
  <div class="row">

    </div>
</section>
```

Under the **row's opening tag**, we're going to add in a new tag called **ng-container**. This is a special container that allows us to hook in things like **for loops** without having to directly create an entire new **div** on our page. Speaking of a **for loop**, this is exactly what we'll use with a property called **\*ngFor**. With this property, we can **iterate** over all our **recipes** and use the **getRecipes function** from **recipe service**.

```
<ng-container *ngFor="let recipe of recipeService.getRecipes()">
```

Now that we have the data that'll show all our recipes, we need to create the front-end look that will actually *show* all the recipes. To do this, we will create **cards** using special **card Bootstrap classes** under our **ng-container**. These cards will be able to use the id, image, title, and description stored with our recipes to dynamically create each card.

In the code below, there are a few special things to take note of in doing this. For our **image src**, we need to encase the src in brackets so it's **[src]**. By doing this, we tell the page that our **src** is **bound** to **javascript** using a special **Angular** feature called **Two-way Data Binding**. For the rest of our property calls, we need to encase them in **double curly braces**. Again, this uses our **binding** to tell the page to directly **render** the **javascript data** on each page. This also applies to our **routerLink** which will use the dynamically retrieved **ID** to know which page it's going to.

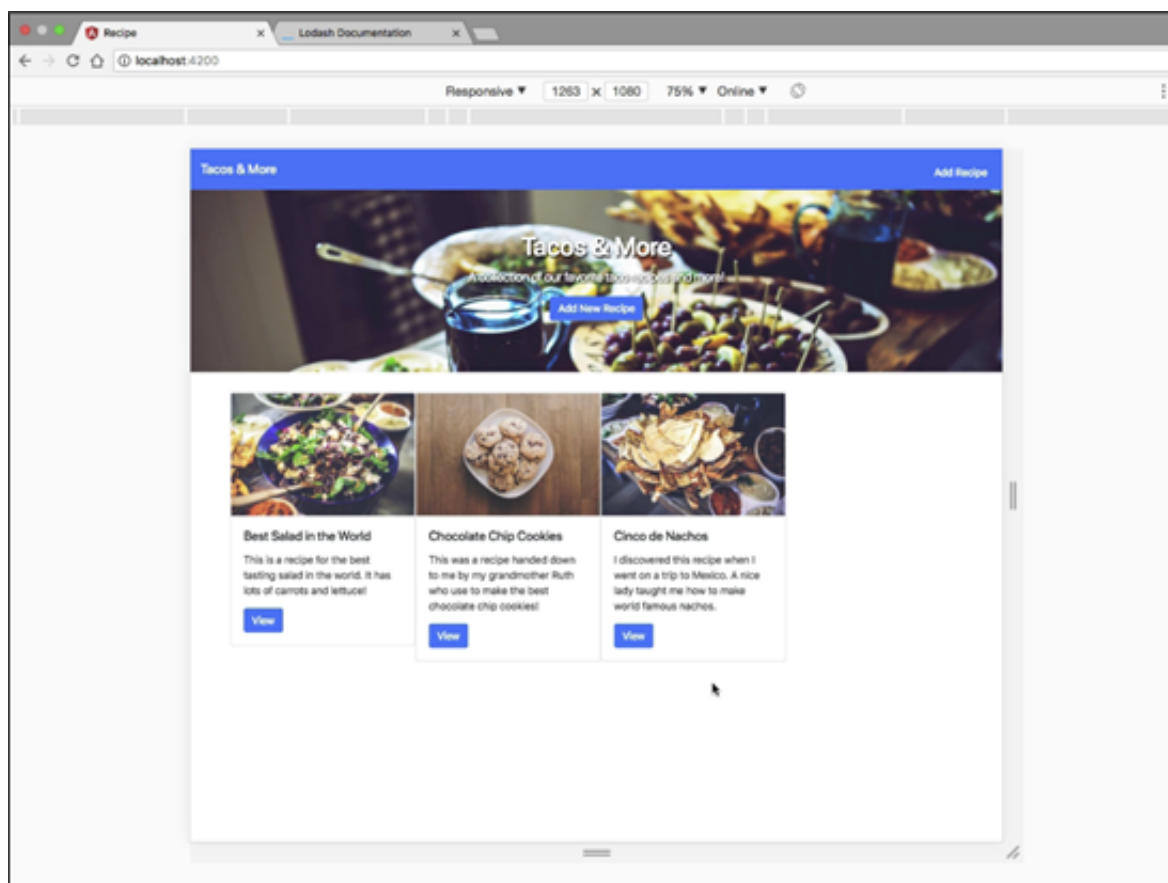
```
<div class="card">
  <img class="card-img-top" [src]="recipe.imageUrl" alt="{{recipe.title}}">
  <div class="card-body">
```

```
<h5 class="card-title">{{recipe.title}}</h5>
<p class="card-text">{{recipe.description}}</p>
<a routerLink="/recipes/{{recipe.id}}" class="btn btn-primary">View</a>
</div>
```

The last thing we want to do is add some styling to our cards. In **home.component.scss**, we're going to specify a width and margin to give our cards a reasonable size.

```
.card {
  margin-bottom: 2em;
  width: 18em;
}
```

If you test your page, you should now successfully see all the cards rendered for our recipes. Thanks to our **for loop** in **ng-container**, each card is created dynamically and is able to access all the data stored in our **JSON** file!



In the next lesson, we'll begin working on showing the individual recipe pages.

If you would like to learn more about the **Bootstrap classes** used in this lesson, please check out the **documentation**: <https://getbootstrap.com/docs/4.3/components/card/>

You can also learn more about displaying data as we did in the **Angular documentation**: <https://angular.io/guide/displaying-data>

In this lesson, we're going to work on the logic that will be utilized to show each recipe for our **show recipe page**.

## The Constructor and Imports

To set up our logic, we'll be working primarily in **show-recipe.component.ts**. Our first step is to set up variables in our pre-provided **constructor**. Of course, like before, we will be using our **recipe service** to be able to use those functions. For this one, we also need to be able to access our router and know which route we're on, as well as a location so we can later create a back button. In our **constructor**, we'll store all this data in variables.

```
constructor(  
  private route: ActivatedRoute,  
  private router: Router,  
  private recipeService: RecipeService,  
  private location: Location  
) { }
```

At the top of the file, you'll notice this auto-generated some **import** code lines for you. On the **import** for **Activated Route**, we need to add the **Router** in addition since they're coming from the same place. While we're here, we'll also manually add an **import** for **Subscription** and an **import** for our **Recipe class**.

```
import { Recipe } from './../../classes/recipe';  
import { RecipeService } from './../../services/recipe.service';  
import { Component, OnInit } from '@angular/core';  
import { ActivatedRoute, Router } from '@angular/router';  
import { Location } from '@angular/common';  
import { Subscription } from 'rxjs';
```

## Using Subscription

In order to do things with our route, we need to utilize **Subscription**. This particular class is able to listen for a specific "event" and then perform the rest of the commands once everything is resolved. Since we need to know our route before we can do anything else, this is perfect for our needs.

Our first step is to add a variable of **subscriptions** in our **ShowRecipeComponent class** above our **constructor**. This allows us to keep track of them, because without removing them later we might cause a memory leak. We'll also create a **recipe variable**, to use both here and in our HTML.

```
private subscriptions: Subscription[] = [];  
public recipe: Recipe;
```

Now we can set up our **subscriptions** in **ngOnInit**, a function that will load right before the page itself renders. Since we set up an **array** for **subscriptions**, we will need to **push** contents into that array. Using **route.paramMap**, we can set up a listener that will wait for our route's URL to determine what to do next.

```
ngOnInit() {
```



```
this.subscriptions.push(  
  this.route.paramMap.subscribe {  
  
  })  
)  
}
```

Remember back in our **recipe service** function we declared the **path** of this **component** to be **recipes/:id**. Using the colon, we declared the **id** as a **parameter**, which our code is able to recognize. Thus, in our listener, we can specifically **get** that **id** and save the value. Then, using our **getRecipeById** function in **recipe service**, we can pass in the **id** that was part of the URL to determine which recipe gets shown. Keep in mind that we have to cast the **id** as a **number** for it to work properly.

```
ngOnInit() {  
  this.subscriptions.push(  
    this.route.paramMap.subscribe(params => {  
      const recipeId = params.get('id');  
      this.recipe = this.recipeService.getRecipeById(parseInt(recipeId));  
    })  
  )  
}
```

## Unsubscribe, Back, and Delete Recipe

We've successfully created our **subscription** logic, but now we need to be able to **unsubscribe** and remove what was pushed into our **subscriptions array**. To do this, we need to use **ngOnDestroy** which is called when the component is being garbage collected. We begin by **importing OnDestroy** from **@angular/core**.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
```

Under the **ngOnInit** function, we'll add our new **ngOnDestroy** function. For this, we merely need to **iterate** over each item in the **array** and **unsubscribe** from it. Even though we only have one element in the array at a time, it is good practice since there are instances where you might have more in other applications.

```
ngOnDestroy() {  
  this.subscriptions.forEach(sub => sub.unsubscribe());  
}
```

Our subscription elements are all set up, but now we need to use our **location variable** to create a **back** function to return to the previous page. Fortunately, we have a built in function to use for this that will tell the router to take us back to our last known location.

```
back() {  
  this.location.back();  
}
```



```
}
```

The last thing we want to do is create a **function** that is able to **delete** a **recipe**. Since our **recipe variable** from earlier already knows the **id**, we can pass that through our **deleteRecipe function**. With the recipe gone, though, we need to be sent back to our home page. We can do this by telling the **router** to **navigate** us back there with our blank path catchall.

```
deleteRecipe(): void {  
  this.recipeService.deleteRecipe(this.recipe.id);  
  this.router.navigate(['']);  
}
```

Our logic is all done, so in the next lesson we can work on our **show component's HTML**.

In this lesson, we're going to work on the **front end** for our **show-recipe component**. Again, we'll be using a lot of **Bootstrap classes** to change widths, colors, use rows, and more. Feel free to check out the **documentation** to learn more about any specific class we're using:

<https://getbootstrap.com/docs/4.3/getting-started/introduction/>

## Top of Page HTML

Head over to **show-recipe.component.html** and remove everything **except** the **navbar**. The first thing we'll do is add an image to the top. Since we already set up the logic, we can use the same method we did for the homepage with **curly braces** to drop in various data (like the image URL). We'll also create a **div container** under it where the rest of our page content will go.

```

<div class="container">

</div>
```

Our first element in our **div container** will be a **delete button** that calls the **delete recipe function** we created logic for. To be able to utilize our logic set up, we need to specify we want it to wait for a **click** and then tell it the function call to perform.

```
<div class="row w-100 d-flex justify-content-end">
  <button class="btn btn-danger" (click)="deleteRecipe()">Delete</button>
</div>
```

Under our **delete button**, we'll create places for our title and description next. **Bootstrap** is able to see **12 columns**, so we'll set these elements to be **six columns wide**. We'll also use **justify-content-center** and **text-center** to center our elements on the page.

```
<div class="row d-flex justify-content-center">
  <div class="col-6">
    <h1 class="text-center">{{recipe.title}}</h1>
    <p class="text-center">{{recipe.description}}</p>
  </div>
</div>
```

So that we can test our page so far, we're going to **show-recipe.component.scss** to alter our image's styling very quickly. Besides altering the width and height, we'll use **object-fit** to make our image fit nicely on the page. Keep in mind, however, that this property doesn't work for **IE 11**.

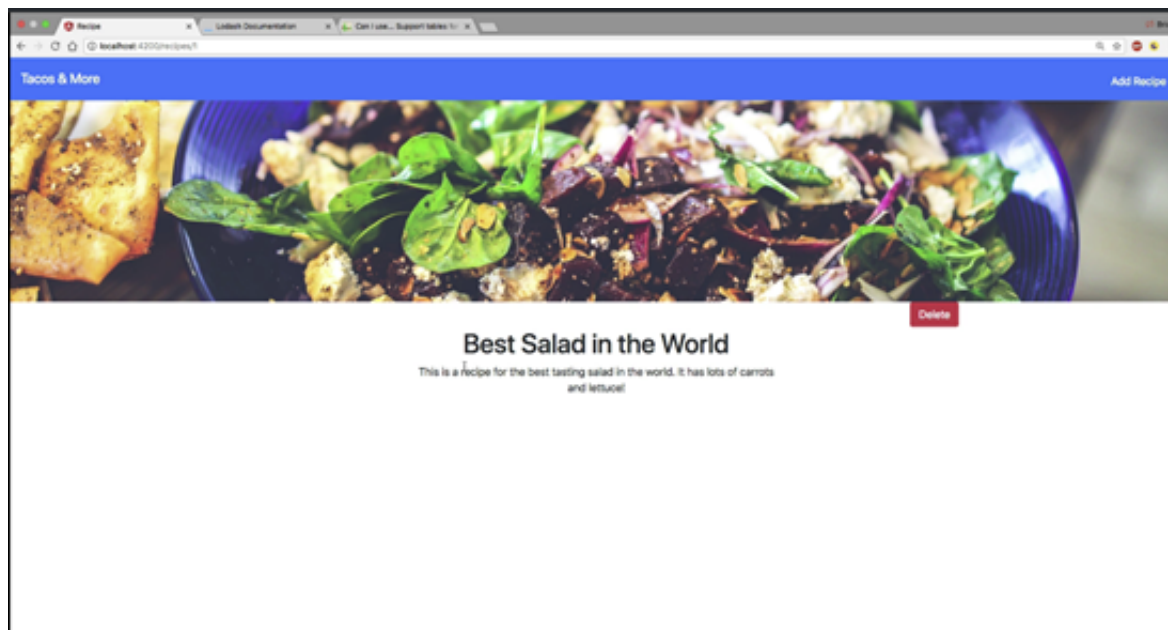
```
.header-image {
  width: 100%;
  height: 300px;
  object-fit: cover;
}
```

(**Note:** You can use sites like <https://caniuse.com/> to check whether a property is compatible with



certain browsers).

So far, our page should look like this:



## Main Content's HTML

The top of the page looks nice, but we need to work on the rest of our content for the recipe. Above the **closing tag** for the **div container**, we'll add another **div** section titled **main**. The first element in **main** that we'll add is the information on how much the recipe serves, which we'll center across the entire **12 columns** on the page.

```
<div class="main">
  <div class="row d-flex justify-content-center">
    <div class="col-12 text-center">
      <h3>Serves: <span class="serves">{{recipe.serves}}</span></h3>
      <hr>
    </div>
  </div>
</div>
```

Above the **main div closing tag**, we're now going to add our **ingredients** and **instructions**. For these, we're going to have **ingredients** take up **four columns** on the left and **instructions** take up **six columns** on the right as an **ordered list**. Remember, though, that both these data elements are a bit special. Thus, in order to use them correctly, we need to use **\*ngFor** to **iterate** through them properly. For **ingredients**, we use this loop to get every pairing of amount and name and display them. In the case of **instructions**, we **iterate** over the **array** of **strings** and display each item separately. Since we do want these **sections** to render, we will put **\*ngFor** directly in the **tags** and **not** in an **ng-container**.

```
<div class="row d-flex justify-content-center">
  <div class="col-4 text-center">
    <h3>Ingredients</h3>
    <div *ngFor="let ingredient of recipe.ingredients">
```



```
        {{ingredient.amount}} - {{ingredient.name}}
    </div>
</div>
<div class="col-6">
    <h3 class="text-center">Instructions</h3>
    <ol>
        <li *ngFor="let instruction of recipe.instructions">
            {{instruction}}
        </li>
    </ol>
</div>
```

Before we can test our **page**, we need to go to our **data.json** and make an alteration. We made a crucial typo, and need to change “instruction” to **instructions** on every recipe. Do that now so your data looks like the following:

```
{
  "recipes": [
    {
      "id": 1,
      "title": "Best Salad in the World",
      "description": "This is a recipe for the best tasting salad in the world. It has lots of carrots and lettuce!",
      "serves": "2 people",
      "imageUrl": "assets/salad.jpg",
      "ingredients": [
        {
          "amount": "1 head",
          "name": "lettuce"
        },
        {
          "amount": "10",
          "name": "carrots"
        },
        {
          "amount": "2 tablespoons",
          "name": "ranch dressing"
        }
      ],
      "instructions": [
        "remove lettuce leaves",
        "add lettuce leaves to a large bowl",
        "then add the carrots the bowl",
        "add the ranch dressing",
        "then mix ingredients together in the bowl"
      ]
    },
    {
      "id": 2,
      "title": "Chocolate Chip Cookies",
      "description": "This was a recipe handed down to me by my grandmother Ruth who use to make the best chocolate chip cookies!",
    }
  ]
}
```

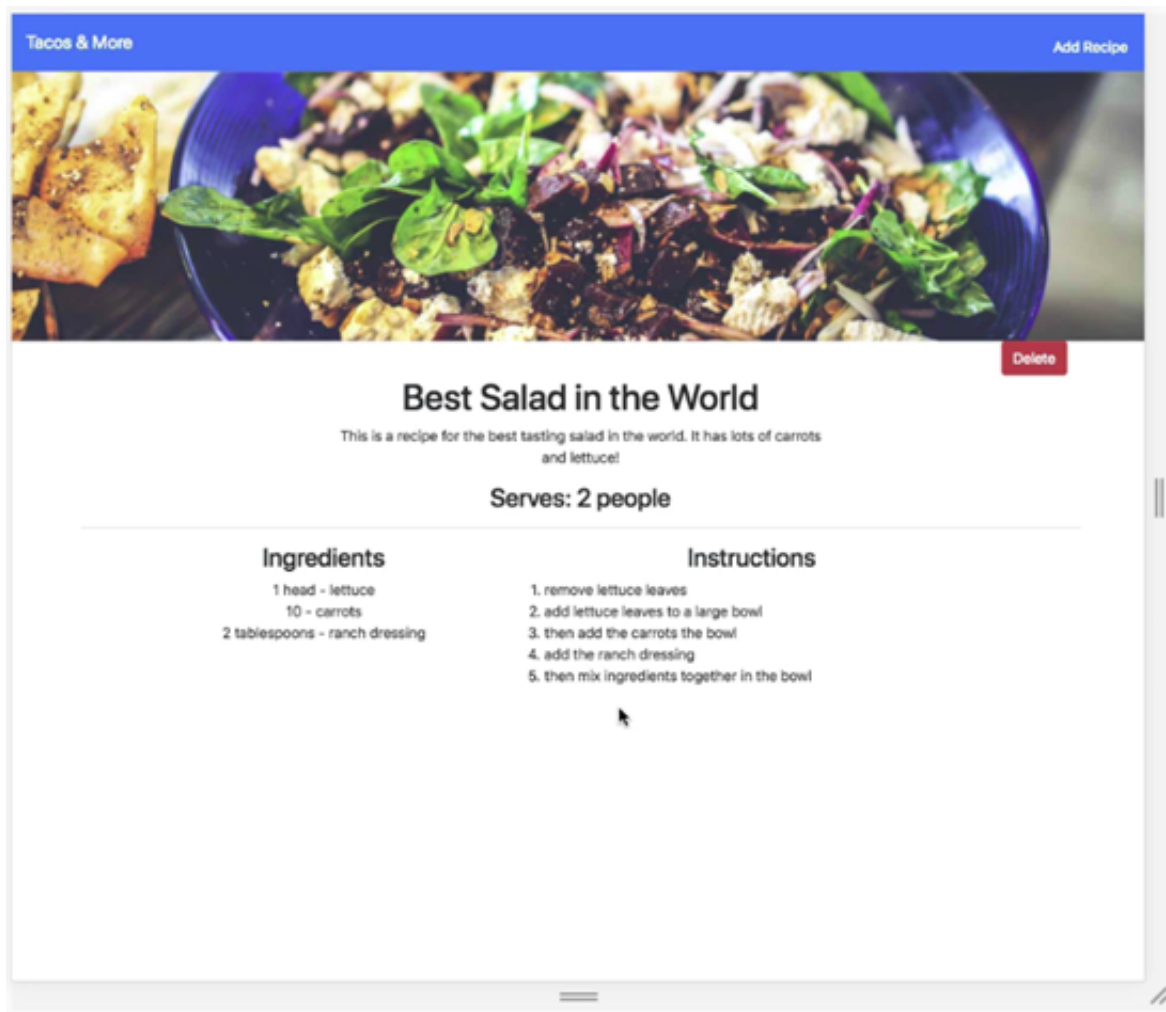


```
"serves": "4 people",
"imageUrl": "assets/cookies.jpg",
"ingredients": [
  {
    "amount": "1 bag",
    "name": "chocolate chips"
  },
  {
    "amount": "2",
    "name": "eggs"
  },
  {
    "amount": "1 cup",
    "name": "flour"
  },
  {
    "amount": "1 cup",
    "name": "sugar"
  }
],
"instructions": [
  "get a large bowl",
  "crack eggs into the bowl",
  "add the rest of the ingredients to the bowl",
  "mix ingredients with a whisk",
  "then evenly place small clumps of cookie dough on a baking sheet",
  "place baking sheet in oven for 12 minutes at 350 degrees F"
]
},
{
  "id": 3,
  "title": "Cinco de Nachos",
  "description": "I discovered this recipe when I went on a trip to Mexico. A nice lady taught me how to make world famous nachos.",
  "serves": "2 adults or 4 kids",
  "imageUrl": "assets/nachos.jpg",
  "ingredients": [
    {
      "amount": "1 bag",
      "name": "tortilla chips"
    },
    {
      "amount": "1 pound",
      "name": "shredded cheese"
    },
    {
      "amount": "1 can",
      "name": "refried beans"
    }
  ],
  "instructions": [
    "spread tortilla chips on a large plate",
    "warm up beans in a pot on the stove",
    "sprinkle the shredded chesse on top of the chips",
    "place plate in the oven for 10 minutes at 350 degrees F",
```



```
"finally remove plate from oven and spread refriend beans on top of the chips"
    ]
  }
]
}
```

Now on our page everything should render fine!



In the next lesson, we're going to style this page just a little bit to make it look better.

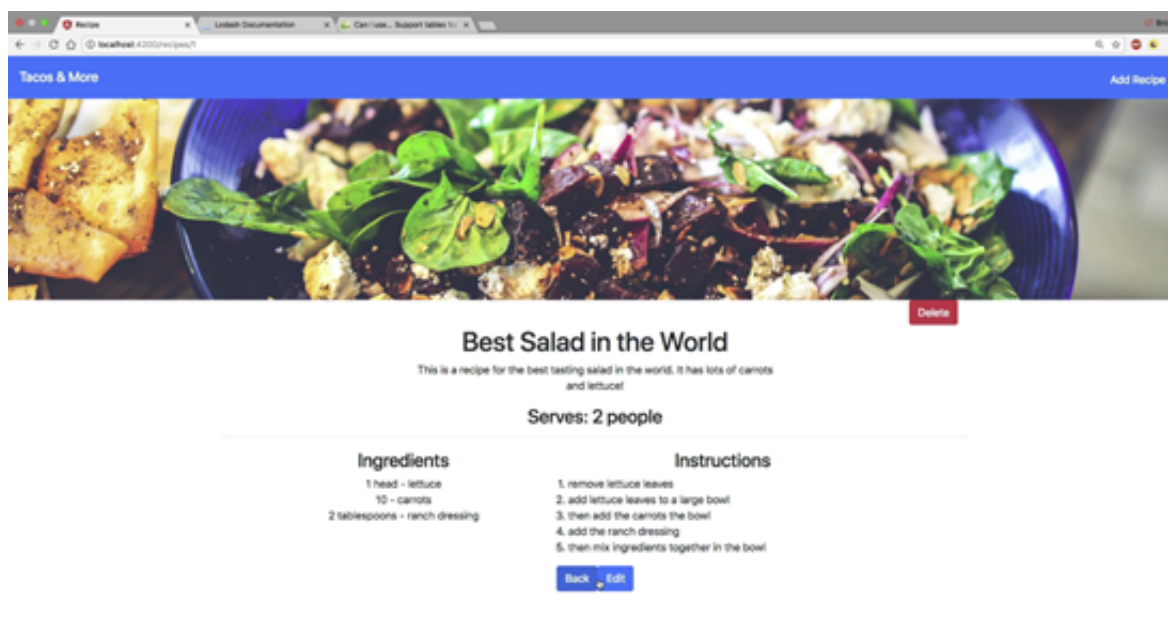
In this lesson, we're going to add some styling to our **show recipe page** and add a few more things.

## Adding Two More Buttons

The first thing we want to do is add two more buttons to our **show page**. Our first button will be our **back** button which uses the **back function** we created several lessons ago in **show-recipe.component.ts**. The second button will be a button to **edit** the recipe. Of course, we need to use **routerLink** again to go to the correct page. Additionally, like before, since **recipe** is storing the recipe's data, we can get the correct **id** via that.

We will add these button right before our **closing tag** for the **div container**.

```
<div class="row back-and-edit d-flex justify-content-center">
  <button class="btn btn-primary" (click)="back()">Back</button>
  <button class="btn btn-
primary" routerLink="/recipes/edit/{{recipe.id}}">Edit</button>
</div>
```



## Styling the Show Page

Now we can quickly style our show page in **show-recipe.component.scss**. With the code below, we'll add in several changes to the **margins** for the headlines, buttons, and various containers we created. So our **buttons** are more consistent, we'll give them a **min-width** of **100px**. We'll also add an **underline** to our **h1** header and give the content for our **main div** a **background color** so it stands out a bit more.

```
h1 {
  margin-top: 1em;
  text-decoration: underline;
}

h2 {
  margin-bottom: 1em;
}
```



```
}

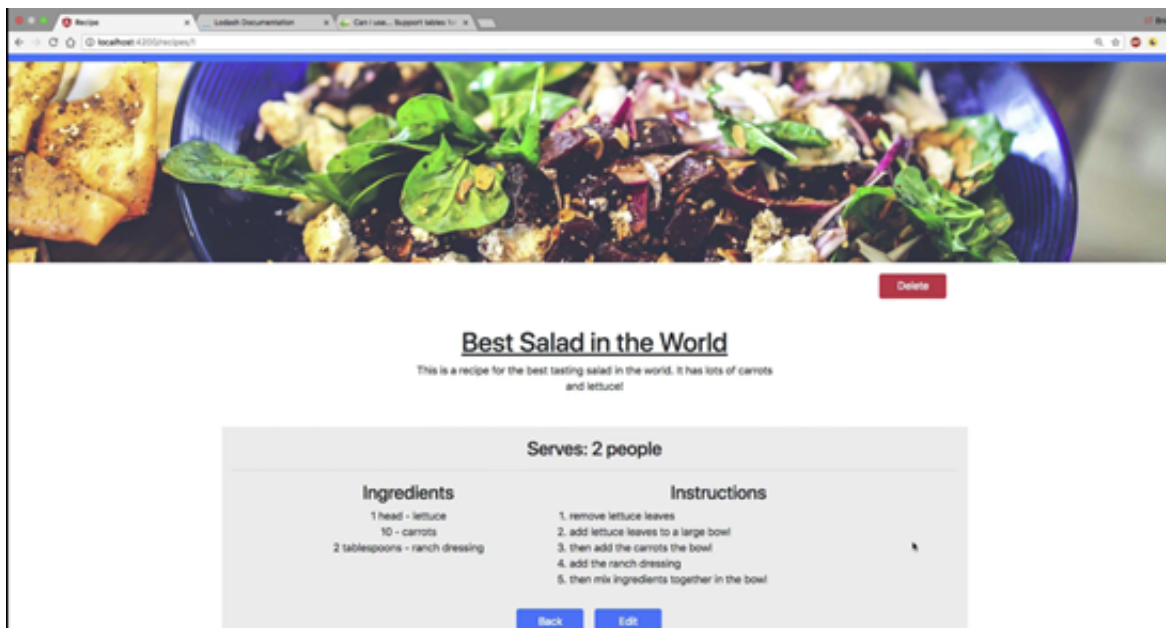
.main {
  background-color: #ececec;
  padding: 1em;
  margin-top: 2em;
}

button {
  margin-top: 1em;
  margin-right: 1em;
  min-width: 100px;
}

.back-and-edit {
  margin-bottom: 2em;
}

.container {
  margin-bottom: 2em;
}
```

Everything on our page is now spaced better and has a better appearance.



You are free to play around with the styling and make it look how you want!

In the next lesson, we'll work on creating the logic for the **edit recipe** page.

In this lesson, we're going to begin setting up the logic that will be used for our **edit recipe page**.

## Importing Important Elements

Now that we can see the recipe, we need to be able to **edit** that recipe. Like before, we will start everything off in our **TypeScript** file, in this case **edit-recipe.component.ts**. Similar to our **show page**, we need to first **import** several of the same route and service logic elements, including **Subscription** since we will be using that again. However, for editing, we will be using **forms** to accomplish the task. As such, we need to import form related materials from **@angular/forms**.

```
import { Recipe } from '../../../classes/recipe';
import { RecipeService } from '../../../services/recipe.service';
import { Location } from '@angular/common';
import { ActivatedRoute, Router } from '@angular/router';
import { Subscription } from 'rxjs';
import { Component, OnInit, OnDestroy } from '@angular/core';
import { FormArray, FormBuilder, FormGroup, Validators } from '@angular/forms';
```

(**Note:** Make sure the **Recipe** import is the class and not the interface.)

In order for our forms to work, we need to **import** the correct **module** for it to the app in general. Head over to **app.module.ts** and add the following to your list of **imports**:

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

Scroll down and also add it into the **@NgModule imports** as well.

```
imports: [
  BrowserModule,
  AppRoutingModule,
  ReactiveFormsModule
],
```

Keep in mind that we're putting all our logic in the **TypeScript** file, so we want the **ReactiveFormsModule**. While there is a **FormsModule**, this is used for cases where you want to wire up more of the logic in the HTML page.

## Basic and Familiar Functions

Back in **edit-recipe.component.ts**, make sure to change your **export class** for the **component** to have **OnDestroy** before we move further.

```
export class EditRecipeComponent implements OnInit, OnDestroy {
```

Now we can start working on the logic in the **class**. Like before, we need **variables** for our **subscriptions** and the **recipe** that is called by the route. Since **instructions** and **ingredients** are special, we need to create **variables** for them to with the **FormArray** type (which will be explained

later). Last but not least, we want one **public variable** for our entire form with the type of **FormGroup**.

```
private subscriptions: Subscription[] = [];  
private recipe: Recipe;  
private instructions: FormArray;  
private ingredients: FormArray;  
public recipeForm: FormGroup;
```

Since we already know we're using **subscriptions**, we can quickly create our **ngOnDestroy function** under our **ngOnInit function**. It will be the same one we used for the **show page's logic**, so you can even copy and paste the function from there.

```
ngOnDestroy() {  
  this.subscriptions.forEach(sub => sub.unsubscribe());  
}
```

Next, we'll work on the **constructor**. Similarly to before, we will get the **ActivatedRoute, Router, recipeService**, and **Location** as part of our **constructor**. Since we're going to use forms on this page, we also need to create a part in our **constructor** for a **FormBuilder**. We will call this **fb** since this is the most common terminology used for it.

```
constructor(  
  private route: ActivatedRoute,  
  private recipeService: RecipeService,  
  private location: Location,  
  private fb: FormBuilder,  
  private router: Router  
) { }
```

With all this set up, we can move on to creating the **function** in **ngOnInit**. Like before, we're going to use this to **subscribe** to the page, use the **id parameter** to determine the recipe, and so forth. Insofar, this code will look and function identically to the one on the **show page**. However, we're going to add one more line that will call a **createForm** method for us. We'll be working on this method later, but for now we'll create a placeholder for it under our **ngOnInit function**.

```
ngOnInit() {  
  this.subscriptions.push(  
    this.route.paramMap.subscribe(params => {  
      const recipeId = params.get('id');  
      this.recipe = this.recipeService.getRecipeById(parseInt(recipeId));  
      this.createForm();  
    })  
  )  
}  
  
private createForm(): void {  
}
```





Finally, before we finish this lesson, we'll quickly add the same back button we used in the **show page** above our **ngOnDestroy** function.

```
back() {  
    this.location.back();  
}
```

In the next lesson, we'll continue working on the logic for this page and talk a bit about how forms work in **Angular**. If you'd like to get a head start, you can check out the **documentation** for forms: <https://angular.io/guide/reactive-forms>

In this lesson, we're going to continue working on our **edit page's logic** and talk a bit about forms. You can also learn more about the **Reactive Forms module** via the **Angular documentation**: <https://angular.io/guide/reactive-forms>

## Form Groups and Fields

The first thing we need to do is create our **createForm function**. This **function** will build essentially build the form dynamically and set the rules for it. Since we'll be using our **recipeForm variable** in the HTML, we need to utilize that first and foremost so we can reference it later on. Next, we need to set up form fields together as a **group**. Though we will only have a top level group declared here, you can subdivide it and group various different form elements together within a group.

```
private createForm(): void {  
    this.recipeForm = this.fb.group({  
  
    });  
}
```

Now we can set up our form fields in the group. Each recipe property will require its own field. Since we'll be pre-populating it with our data, we can reference each property stored in the **recipe variable** with exceptions to **ingredients** and **instructions**. We also want our first fields to be required, so we need to add an array of **validators**. For our form, we'll only use **required**, but there are many **validators** (like **maxLength**) that can be used for your own projects. These can be very powerful and why we're choosing to use **reactive forms**.

```
private createForm(): void {  
    this.recipeForm = this.fb.group({  
        title: [this.recipe.title, [Validators.required]],  
        description: [this.recipe.description, [Validators.required]],  
        serves: [this.recipe.serves, [Validators.required]],  
        imageUrl: [this.recipe.imageUrl, [Validators.required]]  
    });  
}
```

## Form Groups with Arrays

Last, though, we need fields for **ingredients** and **instructions**. Since these properties are a bit different and have several elements in each one, we need to use an **array** type (which we'll initialize as empty) in order to be able to use them.

```
private createForm(): void {  
    this.recipeForm = this.fb.group({  
        title: [this.recipe.title, [Validators.required]],  
        description: [this.recipe.description, [Validators.required]],  
        serves: [this.recipe.serves, [Validators.required]],  
        imageUrl: [this.recipe.imageUrl, [Validators.required]],  
        instructions: this.fb.array([]),  
        ingredients: this.fb.array([])  
    });  
}
```

Now that our primary group is set up, we need to add some logic in the same function to be able to use those **arrays** for **instructions** and **ingredients**. Essentially, we need to map our recipe's **instructions** and **ingredients** to the form fields we created in the **recipeForm** variable. Since they need to be type **FormArray**, we need to make sure to cast them at that as well.

```
this.instructions = this.recipeForm.get('instructions') as FormArray;  
this.ingredients = this.recipeForm.get('ingredients') as FormArray;
```

While we are using the above syntax, keep in mind the below code is also a valid syntax to make sure the **data type** is correct.

```
this.ingredients = <FormArray> this.recipeForm.get('ingredients');
```

Now that we can read the data, we can set up small, inline functions that **iterate** over **each** element in **instructions** and **ingredients** and **push** them to our **arrays**. Notice, however, we specifically push a separate **function** call for each one.

```
this.recipe.instructions.forEach(instruction => {  
  this.instructions.push(this.createInstruction(instruction));  
});  
  
this.recipe.ingredients.forEach(ingredient => {  
  this.ingredients.push(this.createIngredient(ingredient.amount, ingredient.name));  
});
```

Since we have called **functions** for each one of our **forEach loops**, we now need to create these as well under our **createForm function**. These functions will return a **FormBuilder**, so we likewise need to create **form builder groups** for each one. Similarly to above, these will simply take our properties that we pass in, whether an instruction or the ingredients' amounts and name, to create form fields with our **required Validator**.

```
private createInstruction(step: string): FormGroup {  
  return this.fb.group({  
    step: [step, [Validators.required]]  
  });  
}  
  
private createIngredient(amount: string, name: string): FormGroup {  
  return this.fb.group({  
    amount: [amount, [Validators.required]],  
    name: [name, [Validators.required]]  
  });  
}
```

## Summary



To summarize what we've done, with our **createForm function** we created **form groups** with fields for all our recipe properties. In each of these, we initialized it with its default value that is stored in our **JSON data file**. For **ingredients** and **instructions**, we cast them as **Form Arrays** so that we could take each property's elements and generate **groups** for the fields. Then, we **pushed** these **form fields** to an **array** that our main form builder can then use to construct the form.

In the next lesson, we'll finish up creating the logic for our edit page so we can dynamically add & delete instructions and ingredients without hard coding a limit.



In this lesson, we're going to finish up our **edit page's logic** in **edit-recipe.component.ts**.

## Adding Instructions and Ingredients

The primary thing we want to add this lesson is the ability to dynamically add or delete instructions and ingredients. We'll start with adding them, as this is relatively simple. Like we did in the **createForm** function, we're going to **push** our **createInstruction** or **createIngredient** **function** results to our **FormArrays**. Instead of passing in a value from our **JSON**, though, we're going to pass in empty strings. In this way, fields will be added, but they will be blank. We'll add these functions above our **back function**.

```
addInstruction(): void {  
  this.instructions.push(this.createInstruction(''));  
}  
  
addIngredient(): void {  
  this.ingredients.push(this.createIngredient('', ''));  
}
```

## Deleting an Instruction or Ingredient

Now we can work on creating functions to **delete** an instruction or ingredient. Since our form fields for these two are stored in arrays, we can pass in the **index value** to determine which instruction or ingredient to delete. However, in order to get the right **array** with our stored values, we need to use **controls**. Earlier when we created our form fields in the **group**, we were creating what's known as **controls**. Thus, we need to reference the right **control** in **recipeForm**. Once we do that, we can simply save the reference and **remove** the instruction or ingredient at the passed in **index**. We'll add these under our functions from above.

```
deleteInstruction(index: number): void {  
  const arrayControl = this.recipeForm.controls['instructions'] as FormArray;  
  arrayControl.removeAt(index);  
}  
  
deleteIngredient(index: number): void {  
  const arrayControl = this.recipeForm.controls['ingredients'] as FormArray;  
  arrayControl.removeAt(index);  
}
```

## Submitting the Form

Under our **delete functions**, the last thing we want to add is a **function** to **submit the form**. Since we have required fields for our form, we need to first implement an **if/else statement** that will check if **recipeForm**, which stores all our form data, is **valid**. This is a special function of **Reactive Forms**, which have many more validation checks you can learn about in the **documentation**: <https://angular.io/guide/form-validation#reactive-form-validation>

With that established, in our **if statement**, we need to **destructure** our various data points from the **recipeForm** so each field is seen separately. While **ingredients** are a true object, **instructions** is just an array of strings. In order for it to work for our data, we need to save it in a temporary new **variable** that will **map** the saved values in **instructions** and give us each **step** that

we have for them.

```
submitForm(): void {
  if (this.recipeForm.valid) {
    const {title, description, serves, imageUrl, ingredients, instructions} = this.recipeForm.value;
    const filteredInstructions = instructions.map(item => item.step);

    } else {

    }
}
```

Now that we have all this, we can use the **UpdateRecipe function** from our **recipe service**. For this, all we need to do is pass in a **recipe type object** that contains all our **destructured data** from the form. Keep in mind we don't update the **ID**, as this is needed to determine which recipe is being updated. The last step for our **if statement** is to have the **router navigate** us to the regular show page by the **recipe's id**. Note the syntax used to inject the variable into the path.

```
submitForm(): void {
  if (this.recipeForm.valid) {
    const {title, description, serves, imageUrl, ingredients, instructions} = this.recipeForm.value;
    const filteredInstructions = instructions.map(item => item.step);
    const val = this.recipeService.updateRecipe(
      new Recipe(
        {
          id: this.recipe.id,
          title,
          description,
          serves,
          imageUrl,
          ingredients,
          instructions: filteredInstructions
        }
      )
    );

    this.router.navigate(['`/recipes/${this.recipe.id}`']);
  } else {

  }
}
```

For our **else statement**, all we're going to do is add a **console log** indicating an error, though there are more advanced things you can try on your own.

```
else {
  // else show an alert
  console.log("Form Error");
}
```



Our logic is finally complete, so in the next lesson we can begin building the page's HTML.



In this lesson, we're going to begin working on the HTML portion of our edit page.

## Quick Page Styling

Before we cover the HTML, we're going to head into **edit-recipe.component.scss** and give our page some basic styling. For the most part, we're going to **copy** the same exact styling we used for the **show page**, including the cover image, the headline styling, and the different colored section for main. Unlike the show page, we'll give our form itself a margin and use a **nested property** of **last child** so the last button doesn't have a margin on the right.

```
.header-image {
  width: 100%;
  height: 300px;
  object-fit: cover;
}

h1 {
  margin-top: 1em;
  text-decoration: underline;
}

h2 {
  margin-top: 2em;
  margin-bottom: 1em;
}

.main {
  background-color: #ececce;
  padding: 1em;
  margin-top: 2em;
}

button {
  margin-right: 1em;
  min-width: 100px;

  &:last-child {
    margin-right: 0;
  }
}

form {
  margin-top: 1em;
}

.back-and-save {
  margin-top: 2em;
  margin-bottom: 2em;
}
```

## Starting the HTML

Now we can move on to **edit-recipe.component.html**. You can clear the page again except for the





**navbar.** We'll first begin by adding an image and headline. Again, like before, we know which recipe is being served to us, so we can use our **double curly braces** to dynamically access things like the **image URL**. We of course will include a **div container** to put the rest of our contents in, including the headline. For the headline, we're using an empty **div** of **one column** and then a **six column div** so that the headline appears more centered. Sometimes it is necessary to visually offset elements like this so they look correct.

```
<app-navbar></app-navbar>

<div class="container">
  <div class="row d-flex justify-content-center">
    <div class="col-1"></div>
    <div class="col-6">
      <h1 class="text-center">Edit Recipe</h1>
    </div>
  </div>
</div>

</div>
```

Above the **closing tag** for the **div container**, we're going to start the section for our form. Since we need to use the **recipeForm variable**, we need to use **brackets** to bind a **formGroup** as javascript. We also don't want the HTML itself to validate our form (since our component's logic will handle this), so we also need to specify **novalidate** in the tag.

```
<form [formGroup]="recipeForm" novalidate>

</form>
```

## Recipe Title Field

Let's first create the area for our **recipe title field**. For this, we need a **label** and an **input field**, so we'll encase both in a **form-group div** with various other Bootstrap classes. For our label, we will create a **two column** wide headline that is aligned to the right of its container with **justify-content-end** and vertically aligned center with **align-items-center**.

```
<div class="row">
  <div class="form-group w-100 title d-flex justify-content-center">
    <div class="col-2 d-flex justify-content-end align-items-center">
      <label class="mb-0"><h4 class="mb-0">Title:</h4></label>
    </div>
    <!--Space for our input field-->
  </div>
</div>
```

Under our **two columns** that contain the **label**, we'll add in our input field which will be **six columns** wide. Earlier we **mapped** our form to our **recipeForm**, so we are able to assign our inputs so they correlate to the appropriate field with **formControlName**. Since this is for our title, we use **"title"** to correctly assign it for that property value.



```
<div class="row">
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Title:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Title..." formControlName="title">
  </div>
  <div class="col-1"></div>
</div>
</div>
```

(**Note:** Like before, we're using an empty one column div to align things correctly).

## Other Fields

Fortunately, all the code above is very modular, so we can **copy and paste** it several times for some of our needed fields. With our **pasting**, though, we need to make sure to change the **placeholder** in the **input** to be the correct name as well as the **formControlName**. We also need to change the **label**. We'll do this for **description**, **serves**, and **image URL**, so our code so far will look like the following:

```
<div class="row">
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Title:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Title..." formControlName="title">
  </div>
  <div class="col-1"></div>
</div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Description:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Description..." formControlName="description">
  </div>
  <div class="col-1"></div>
</div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Serves:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Serves..." formControlName="serves">
```



```
</div>
<div class="col-1"></div>
</div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Image Url:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Image Url..." formControlName="imageUrl">
  </div>
  <div class="col-1"></div>
</div>
</div>
</div>
```

In the next lesson, we'll work out the sections for our **instructions** since these will require a bit more work.

If you need any clarification on the **Bootstrap classes** used for this lesson, please check out the **Bootstrap 4 documentation**: <https://getbootstrap.com/docs/4.3/getting-started/introduction/>

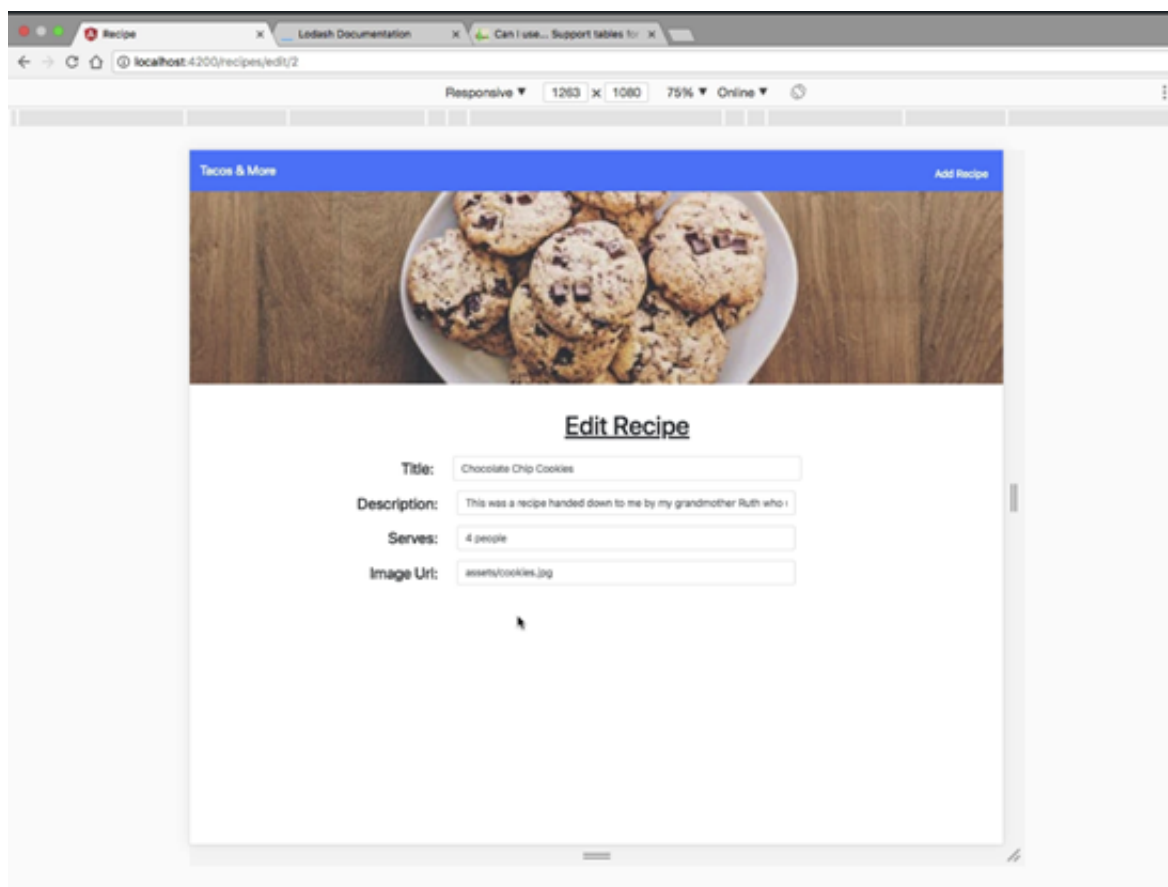
In this lesson, we're going to work on the **instructions** for our edit page's form in **edit-recipe.component.html**.

## Fixing Errors from Last Lesson

Before we begin, we need to fix some errors from last lesson. First and foremost, copying and pasting might have caused **Visual Studio Code** to insert extra **closing div tags** at the end of each section. We need to clean those up so our code doesn't throw any errors. Make sure your fields' code looks like the one below before we move on:

```
<div class="row">
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Title:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Title..." formControlName="title">
  </div>
  <div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Description:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Description..." formControlName="description">
  </div>
  <div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Serves:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Serves..." formControlName="serves">
  </div>
  <div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Image Url:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Image Url..." formControlName="imageUrl">
  </div>
  <div class="col-1"></div>
</div>
</div>
```

If you run **ng serve** and test the page, you can see what we have so far.



## Creating Instruction Fields

Just before the **form closing tag**, we're going to add in our section for **instructions**. Like we did with **formControlName**, we need to declare this section as a **formArrayName**, since that's what instructions consists of.

```
<div formArrayName="instructions">

</div>
```

Within this **div** section, the first thing we'll add is a headline for what the form fields are for. The section will be remarkably similar to the page's headline, including the use of an **empty one column div** and **six column div** for the headline.

```
<div class="row d-flex justify-content-center">
  <div class="col-1"></div>
  <div class="col-6">
    <h2 class="text-center">Instructions</h2>
  </div>
</div>
```

Under the previous code, we now need to get the **controls** that are in our **formArray**. To do this, we need to **loop** through the **controls** for the **instructions array** using **\*ngFor**. We're also going to add a variable to our loop so we can store the **index**.

```
<div *ngFor="let control of recipeForm.get('instructions').controls; let i = index">

</div>
```

Inside this section for **\*ngFor**, we're going to make another **div**. This one will have **[formGroupName]** as a property that's connected to our **index variable**.

```
<div [formGroupName]="i" class="form-group w-100 title d-flex justify-content-
center">

</div>
```

Now we can move on to the actual **label** and **input** field inside this last **div**. Fortunately, this section is similar to the sections we did last lesson, including use **formControlName** to correctly connect the right **control name** we set up (in this case **step**). However, since we're storing our **index**, we can use it to set the **label** dynamically by **adding 1 to the index**. This will number the steps for us appropriately!

```
<div class="col-2 d-flex justify-content-end align-items-center">
  <label class="mb-0"><h4 class="mb-0">{{i+1}}:</h4></label>
</div>
<div class="col-6 d-flex align-items-center">
  <input type="text" class="form-
control w-100" placeholder="add an instruction..." formControlName="step">
</div>
```

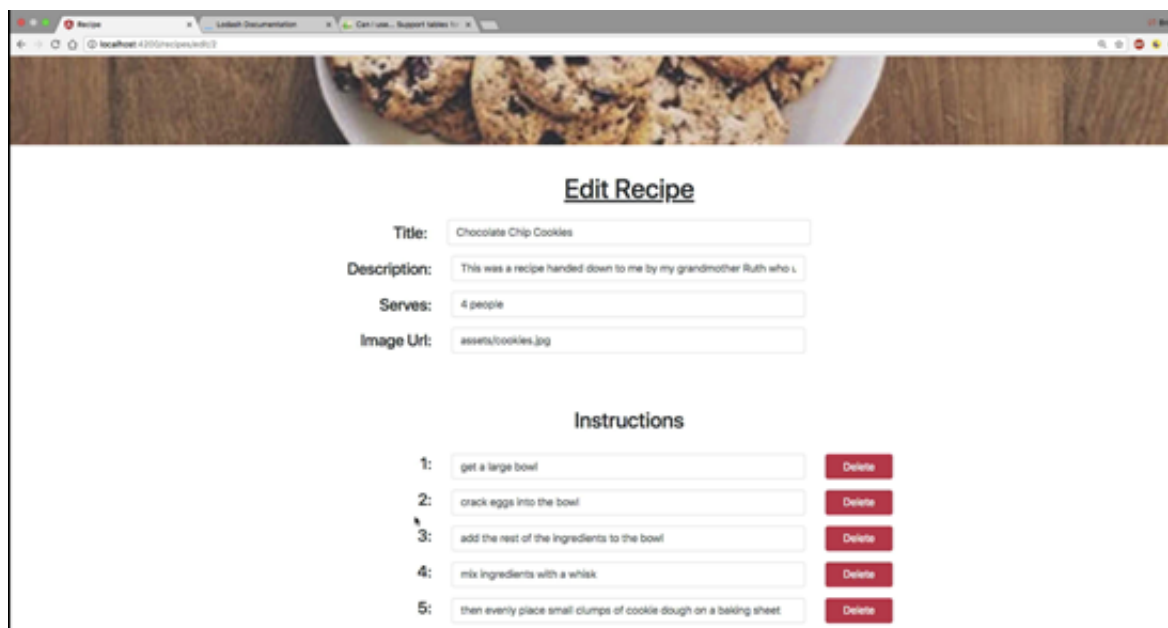
(**Note:** Javascript arrays start at 0 for the index.)

## Delete and Add Instruction Button

Next, we need to add in a button that allows us to delete the **instruction**. Before the **closing tag** for the **div** section with **formGroupName**, we'll add another **div** that contains our button. Similarly to a previous lesson, we need to use (**click**) to wait for the button to be clicked. Then, with that click, we can use our **deleteInstruction function** by passing in the **index** that we saved earlier once again.

```
<div class="col-1 d-flex align-items-center">
  <button class="btn btn-danger" (click)="deleteInstruction(i)">Delete</button>
</div>
```

We can test it and see everything is working as intended.



The screenshot shows a web browser window with the URL `localhost:4200/recipes/edit/2`. The page displays an "Edit Recipe" form. At the top is a header image of cookies. The form fields are:

- Title:**
- Description:**
- Serves:**
- Image Url:**

Below the form is a section titled "Instructions" with a list of five items, each with a "Delete" button:

- 1:  Delete
- 2:  Delete
- 3:  Delete
- 4:  Delete
- 5:  Delete

The last thing we need to add is a button to add instructions. Before the **closing tag** for the **div** with the **formArrayName** for **instructions**, we will add this button. Similar to delete, we'll wait for a **click** and then use our **addInstructions function** (which takes no parameters) to add a field.

```
<div class="w-100 d-flex justify-content-center">
  <div class="col-1"></div>
  <div class="col-6">
    <button class="btn btn-success w-100" (click)="addInstruction()">Add</button>
  </div>
</div>
```

In the next lesson, we will finish up our **edit page's** HTML and create a section for our **ingredients**.

Remember that if you need clarification on any **Bootstrap classes** we used, you can view the **documentation**: <https://getbootstrap.com/docs/4.3/getting-started/introduction/>



In this lesson, we're going to add in the **ingredients** section for our **edit page** in **edit-recipe.component.html**. Be sure to refer to the **Bootstrap 4 documentation** for any needed information on the **Bootstrap classes** used: <https://getbootstrap.com/docs/4.3/getting-started/introduction/>

## The Ingredients Fields

Since most of our code will be the same, we can save ourselves time by **copying** the entire **div formArrayName** for **instructions** and **pasting** it above our **closing form tag**. Then, for the most part, we just need to **change** every mention of instructions to **ingredients**.

```
<div formArrayName="ingredients">
  <div class="row d-flex justify-content-center">
    <div class="col-1"></div>
    <div class="col-6">
      <h2 class="text-center">Ingredients</h2>
    </div>
  </div>
  <div *ngFor="let control of recipeForm.get('ingredients').controls; let i = index">
    <div [formGroupName]="i" class="form-group w-100 title d-flex justify-content-center">
      <div class="col-2 d-flex justify-content-end align-items-center">
        <label class="mb-0"><h4 class="mb-0">{{i+1}}:</h4></label>
      </div>
      <div class="col-6 d-flex align-items-center">
        <input type="text" class="form-control w-100" placeholder="add an ingredient..." formControlName="step">
      </div>
      <div class="col-1 d-flex align-items-center">
        <button class="btn btn-danger" (click)="deleteIngredient(i)">Delete</button>
      </div>
    </div>
  </div>
  <div class="w-100 d-flex justify-content-center">
    <div class="col-1"></div>
    <div class="col-6">
      <button class="btn btn-success w-100" (click)="addIngredient()">Add</button>
    </div>
  </div>
</div>
```



The screenshot shows a web application interface with two main sections: 'Instructions' and 'Ingredients'. The 'Instructions' section contains a list of 6 steps, each with a text input field and a 'Delete' button. A green 'Add' button is located below the instructions list. The 'Ingredients' section contains a list of 4 items, each with a text input field and a 'Delete' button.

While it looks functional, remember that our **ingredients** have both an amount and name. Thus, we need to change our actual **inputs** to reflect that. Locate the code below and **remove** it, as we'll be adding new code in its place. Make sure you are in the section for **ingredients**, though!

```
<div class="col-6 d-flex align-items-center">
  <input type="text" class="form-
control w-100" placeholder="add an ingredient..." formControlName="step">
</div>
```

Now we can replace our input fields with the correct ones. Since we will have two side by side, we'll make each input **three columns** wide. Remember that our **ingredients** consist of an **amount** and **name**, so we need to set up our **formControlNames** and **placeholders** to be reflective of that. Otherwise, our fields will still look identical to previous ones we have done.

```
<div class="col-3">
  <input type="text" class="form-
control w-100" placeholder="amount..." formControlName="amount">
</div>
<div class="col-3">
  <input type="text" class="form-
control w-100" placeholder="name..." formControlName="name">
</div>
```

Testing the page, we can see our **ingredients** now look correct!

The screenshot shows a web browser displaying a recipe management application. The page has a title bar with several tabs open. The main content area is divided into two sections: 'Instructions' and 'Ingredients'. The 'Instructions' section contains a list of 6 steps, each with a numbered input field and a 'Delete' button. The 'Ingredients' section contains a list of 4 items, each with a numbered input field, a text input field for the ingredient name, and a 'Delete' button. A green 'Add' button is located below the instructions section.

## Back and Form Submission

Now that all our fields are set up, we can add buttons for our back and save. We will add these right before the **closing tag** for the **form**. Since we already created **functions** for them, all we need to do is use **click** and then connect it to each function where appropriate.

```
<div class="row d-flex justify-content-center back-and-save">
  <div class="col-1"></div>
  <div class="col-6 d-flex justify-content-center">
    <button class="btn btn-primary" (click)="back()">Back</button>
    <button class="btn btn-primary" (click)="submitForm()">Save</button>
  </div>
</div>
```

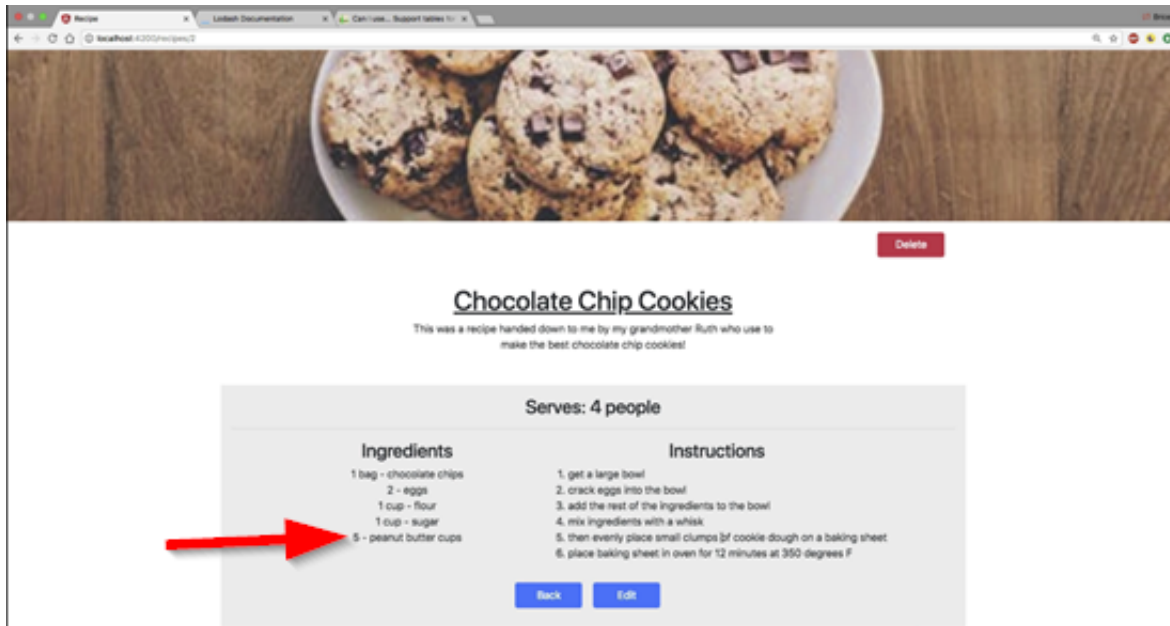
Last but not least, we're going to make a slight adjustment. In the top area before the section for instructions, locate the **div** with just a **class** of **row**. We're going to remove it as this winds up looking tacky. Make sure to remove the **closing tag** for that **div** as well. You can check the code below to verify yours matches.

```
<form [formGroup]="recipeForm" novalidate>
  <div class="form-group w-100 title d-flex justify-content-center">
    <div class="col-2 d-flex justify-content-end align-items-center">
      <label class="mb-0"><h4 class="mb-0">Title:</h4></label>
    </div>
    <div class="col-6">
      <input type="text" class="form-
control w-100" placeholder="Title..." formControlName="title">
    </div>
    <div class="col-1"></div>
  </div>
  <div class="form-group w-100 title d-flex justify-content-center">
    <div class="col-2 d-flex justify-content-end align-items-center">
      <label class="mb-0"><h4 class="mb-0">Description:</h4></label>
```



```
</div>
<div class="col-6">
  <input type="text" class="form-
control w-100" placeholder="Description..." formControlName="description">
</div>
<div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Serves:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Serves..." formControlName="serves">
  </div>
  <div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Image Url:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Image Url..." formControlName="imageUrl">
  </div>
  <div class="col-1"></div>
</div>
<div formArrayName="instructions">
<!--The rest of the code below here should be fine!-->
```

Our edit page is now complete! You can test out the functionality for yourself to ensure that everything is working.



Keep in mind that we're not **saving** data to our **JSON**, so refreshing the page will reset everything.

In the next and final lesson, we will create the **new recipe** page, which will reuse a lot of the logic and HTML we did for this page.

In this final lesson, we're going to create the **new recipe** page for our webapp.

## Quick Fix

Before we start, we want to change a line in our **edit-recipe.component.ts** code. In our **submitForm function**, we saved the value when we created the recipe from the submission. However, we don't really need to do this, so make sure the following three lines read as such:

```
const {title, description, serves, imageUrl, ingredients, instructions} = this.recipeForm.value;
const filteredInstructions = instructions.map(item => item.step);
this.recipeService.createRecipe({
```

## New Recipe Page Logic

We can now make the logic for our new recipe page! Fortunately, most of the logic will be the exact same as we did for the **edit page**. While normally we could refactor everything into a shared component for more efficiency, we're going to simply **copy** and **paste** everything from **edit-recipe.component.ts** for the sake of time. Once you **copy** the **edit-recipe.component.ts** code, **paste** it into **new-recipe.component.ts**.

However, there are a few things we need to change in **new-recipe.component.ts** to make everything work. In the **@Component** area, we need to **change** the word "edit" to be **new** on all three lines so everything matches the correct name for the component. We also need to change the **export class** name to be **NewRecipeComponent** as well.

```
@Component({
  selector: 'app-new-recipe',
  templateUrl: './new-recipe.component.html',
  styleUrls: ['./new-recipe.component.scss']
})
export class NewRecipeComponent implements OnInit, OnDestroy {
  //Rest of code
```

For the **submitForm function**, we need to change the **recipe service** we use to be **createRecipe** instead of **updateRecipe**. Due to this, we have to pass in regular parameters instead of an object. Remember that our **id** will be auto generated, so we don't need to include it. However, because the generation doesn't happen till later, we need to change our **router** to **navigate** to the home page.

```
submitForm(): void {
  if (this.recipeForm.valid) {
    const {title, description, serves, imageUrl, ingredients, instructions} = this.recipeForm.value;
    const filteredInstructions = instructions.map(item => item.step);
    this.recipeService.createRecipe(
      title,
      description,
      serves,
      imageUrl,
      ingredients,
```



```
        filteredInstructions
    );

    this.router.navigate(['']);
  } else {
    // else show an alert
    console.log("Form Error");
  }
}
```

The last and most important change needs to happen in our **createForm function**, however. In this function, we're trying to load in default values for our **FormBuilder group**. Since this new recipe has no default values, we need to load in **empty strings** instead of referencing the values. Similarly, we don't want to iterate over all our **instructions** or **ingredients**, so we will simply change the very bottom of this function to call functions.

```
private createForm(): void {
  this.recipeForm = this.fb.group({
    title: ['', [Validators.required]],
    description: ['', [Validators.required]],
    serves: ['', [Validators.required]],
    imageUrl: ['', [Validators.required]],
    instructions: this.fb.array([]),
    ingredients: this.fb.array([])
  });

  this.instructions = this.recipeForm.get('instructions') as FormArray;
  this.ingredients = this.recipeForm.get('ingredients') as FormArray;

  this.addInstruction();
  this.addIngredient();
}
```

## New Recipe Page HTML

That's all we need for our logic, so we can now create our **new recipe page** HTML. Once again we're going to be using similar code, so we can simply just **copy and paste** from our edit page. First, go into **edit-recipe.component.scss** and **copy and paste** that into **new-recipe.component.scss**. The SCSS file should look like the code below now:

```
.header-image {
  width: 100%;
  height: 300px;
  object-fit: cover;
}

h1 {
  margin-top: 1em;
  text-decoration: underline;
}
```



```
h2 {
  margin-top: 2em;
  margin-bottom: 1em;
}

.main {
  background-color: #ececce;
  padding: 1em;
  margin-top: 2em;
}

button {
  margin-right: 1em;
  min-width: 100px;

  &:last-child {
    margin-right: 0;
  }
}

form {
  margin-top: 1em;
}

.back-and-save {
  margin-top: 2em;
  margin-bottom: 2em;
}
```

Second, **copy** everything from **edit-recipe.component.html** and **paste** it into **new-recipe.component.html**. Quickly, we need to make a few tweaks. Since our new recipe wouldn't have an image yet, we need to remove the image line. Further, we also need to change the page **headline** to be **New Recipe**.

```
<app-navbar></app-navbar>
<div class="container">
  <div class="row d-flex justify-content-center">
    <div class="col-1"></div>
    <div class="col-6">
      <h1 class="text-center">New Recipe</h1>
    </div>
  </div>
  <form [formGroup]="recipeForm" novalidate>
    <div class="form-group w-100 title d-flex justify-content-center">
      <div class="col-2 d-flex justify-content-end align-items-center">
        <label class="mb-0"><h4 class="mb-0">Title:</h4></label>
      </div>
      <div class="col-6">
        <input type="text" class="form-
control w-100" placeholder="Title..." formControlName="title">
      </div>
      <div class="col-1"></div>
    </div>
```



```

<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Description:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Description..." formControlName="description">
  </div>
  <div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Serves:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Serves..." formControlName="serves">
  </div>
  <div class="col-1"></div>
</div>
<div class="form-group w-100 title d-flex justify-content-center">
  <div class="col-2 d-flex justify-content-end align-items-center">
    <label class="mb-0"><h4 class="mb-0">Image Url:</h4></label>
  </div>
  <div class="col-6">
    <input type="text" class="form-
control w-100" placeholder="Image Url..." formControlName="imageUrl">
  </div>
  <div class="col-1"></div>
</div>
<div formArrayName="instructions">
  <div class="row d-flex justify-content-center">
    <div class="col-1"></div>
    <div class="col-6">
      <h2 class="text-center">Instructions</h2>
    </div>
  </div>
  <div *ngFor="let control of recipeForm.get('instructions').controls; let i = in
dex">
    <div [formGroupName]="i" class="form-group w-100 title d-flex justify-content-
center">
      <div class="col-2 d-flex justify-content-end align-items-center">
        <label class="mb-0"><h4 class="mb-0">{{i+1}}:</h4></label>
      </div>
      <div class="col-6 d-flex align-items-center">
        <input type="text" class="form-
control w-100" placeholder="add an instruction..." formControlName="step">
      </div>
      <div class="col-1 d-flex align-items-center">
        <button class="btn btn-
danger" (click)="deleteInstruction(i)">Delete</button>
      </div>
    </div>
  </div>
</div>
<div class="w-100 d-flex justify-content-center">

```





```
<div class="col-1"></div>
<div class="col-6">
  <button class="btn btn-
success w-100" (click)="addInstruction()">Add</button>
</div>
</div>
</div>
<div formArrayName="ingredients">
  <div class="row d-flex justify-content-center">
    <div class="col-1"></div>
    <div class="col-6">
      <h2 class="text-center">Ingredients</h2>
    </div>
  </div>
  <div *ngFor="let control of recipeForm.get('ingredients').controls; let i = ind
ex">
    <div [formGroupName]="i" class="form-group w-100 title d-flex justify-content-
center">
      <div class="col-2 d-flex justify-content-end align-items-center">
        <label class="mb-0"><h4 class="mb-0">{{i+1}}:</h4></label>
      </div>
      <div class="col-3">
        <input type="text" class="form-
control w-100" placeholder="amount..." formControlName="amount">
      </div>
      <div class="col-3">
        <input type="text" class="form-
control w-100" placeholder="name..." formControlName="name">
      </div>
      <div class="col-1 d-flex align-items-center">
        <button class="btn btn-
danger" (click)="deleteIngredient(i)">Delete</button>
      </div>
    </div>
    <div class="w-100 d-flex justify-content-center">
      <div class="col-1"></div>
      <div class="col-6">
        <button class="btn btn-
success w-100" (click)="addIngredient()">Add</button>
      </div>
    </div>
  </div>
  <div class="row d-flex justify-content-center back-and-save">
    <div class="col-1"></div>
    <div class="col-6 d-flex justify-content-center">
      <button class="btn btn-primary" (click)="back()">Back</button>
      <button class="btn btn-primary" (click)="submitForm()">Save</button>
    </div>
  </div>
</form>
</div>
```

Everything is complete already, so you can play around with adding a new recipe!

Tacos & More Add Recipe

### New Recipe

Title:

Description:

Serves:

Image Uri:

#### Instructions

1:  Delete

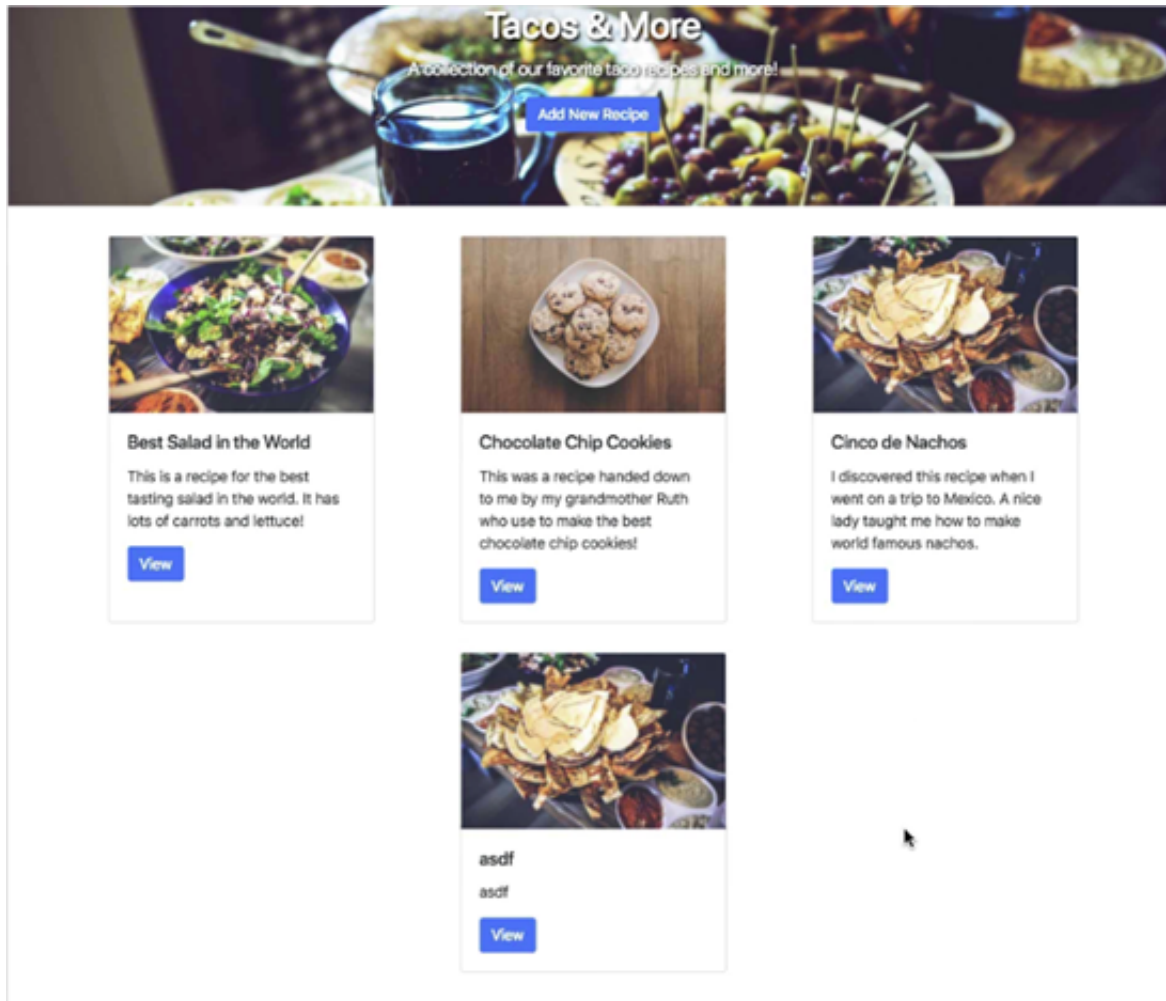
Add

#### Ingredients

1:   Delete

Add

Back Save



**Congratulations!** You've now completed this **Zenva** course on using **Angular** and created your very own web app!

Although improvements could be made, you now have the basics for using **CRUD** in **Angular** and can make your own apps. Be sure to check out the **Angular 7 documentation** for further information on using **Angular**: <https://angular.io/docs>