Front End Building Basics

Exercise Guide

#### Contents

[Prerequisites 3](#_Toc100068126)

[Tutorial 4](#_Toc100068127)

[Step 1: Create a new Angular project 4](#_Toc100068128)

[Step 2: Create a New Todos Component and Using Bootstrap for CSS Framework 7](#_Toc100068129)

[Step 3: Building a To-do List Component 11](#_Toc100068130)

[Step 4: Setting Up the Backend Services 13](#_Toc100068131)

[Step 5: Showing the Data Retrieved on the Website 19](#_Toc100068132)

[Step 6: Adding More Functionalities to our To-do List 22](#_Toc100068133)

[Step 7: Testing and Unit Tests 29](#_Toc100068134)

#### Overview

The purpose of this lab is to create a simple website using Front End tools. We will use Angular as our front-end application framework.

# Prerequisites

Mandatory:

* Text editor (Recommended is [Visual Studio Code](https://code.visualstudio.com/download))
* To install Angular on your local system, you need the following:
* Node.js Installation: <https://nodejs.org/en/>

Node.js is an open source, cross-platform, JavaScript runtime environment that runs on the V8 engine and executes JavaScript / TypeScript code outside a web-browser

* + @angular/cli
  + tslint
  + typescript

*note: to install global packages, after Node JS installation, open your terminal and run:*

npm install -g @angular/cli tslint typescript

For Mac Users:

sudo npm install -g @angular/cli tslint typescript

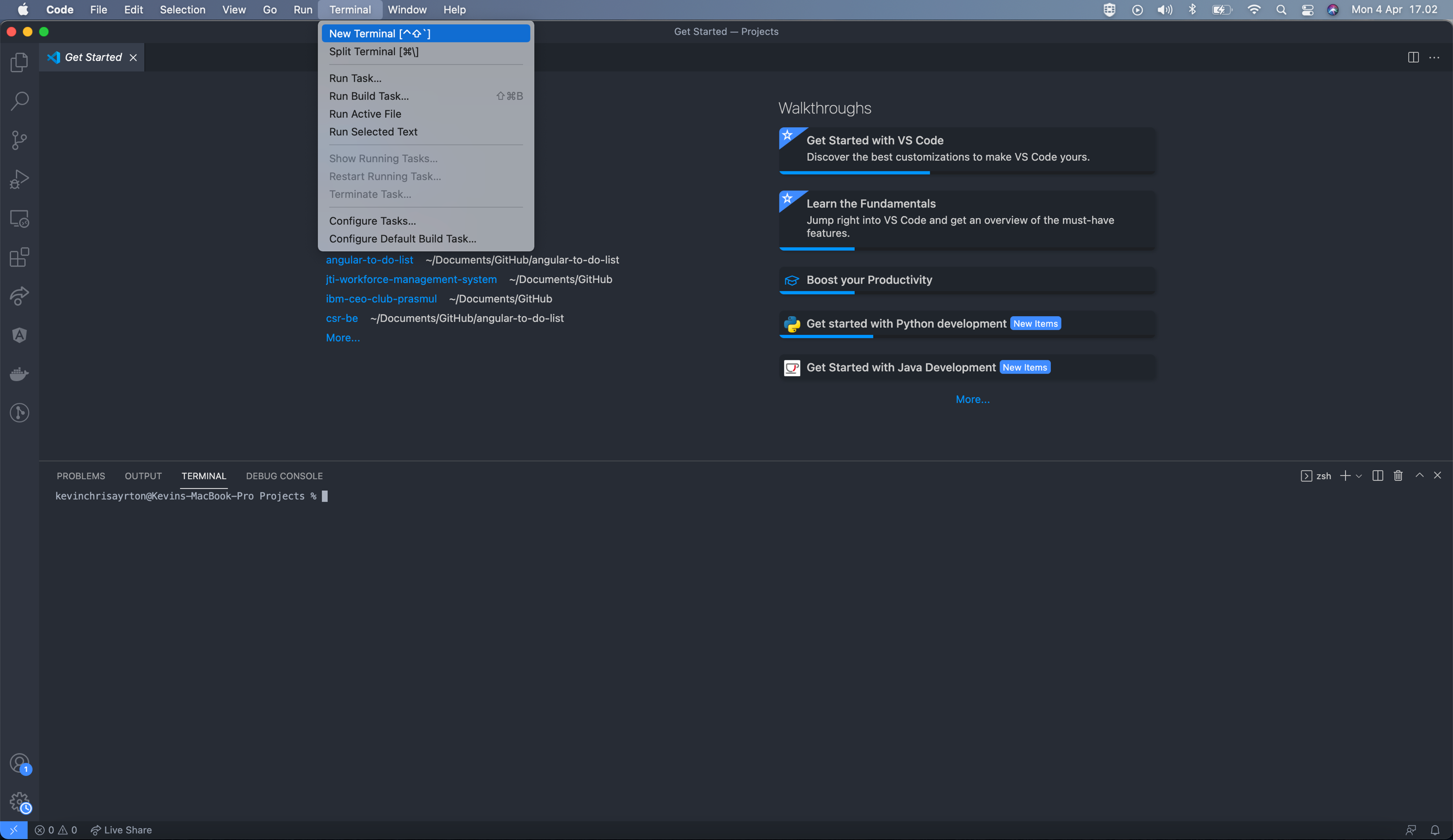
# Tutorial

## Step 1: Create a new Angular project

1. Create a new folder for your project (skip if you already have an existing one from previous classes)

Graphical user interface, text, application

Description automatically generated

1. Open your Visual Studio Code / other text editors you have
2. On the left-hand side tab, click the top menu (Explorer) and click Open Folder
3. Locate the folder that you just made, click on it and press open
4. Open a new terminal
5. Create a workspace and initial application. You develops apps in the context of an Angular workspace.

To create a new workspace and initial starter app:

1. Run the CLI command ng new and provide the name angular-to-do-list, as shown here:

**ng new angular-to-do-list**

For Mac Users (to give permission access):

**sudo ng new angular-to-do-list**

The ng new command prompts you for information about features to include in the initial app.

Accept the include routing by pressing the Enter or Return key and choose the scss by pressing down key and Enter or Return key.

The Angular CLI installs the necessary Angular npm packages and other dependencies.

This can take a few minutes.

The CLI creates a new workspace and a simple Welcome app, ready to run.

1. Now, you have created your first Angular starter application. The Angular CLI includes a server, for you to build and serve your app locally. You can try to build and serve it using this command:

**cd angular-to-do-list**

**ng serve --open**

1. If your installation and setup was successful, you should see a page similar to the following.

Graphical user interface

Description automatically generated

## Step 2: Create a New Todos Component and Using Bootstrap for CSS Framework

1. Open a new terminal, inside the terminal, type in:

**cd angular-to-do-list**

**npm install**

This will let your npm package manager install all the dependencies and packages you needed to build the application.

1. In the file explorer, open the app component, right click and make a new folder called components. This is where we are going to store our components.
2. Go to your terminal, then navigate to components folder:

**cd src/app/components**

And run this command to generate your first Angular component:

**ng g component todos**

This will create you a new todos component and update your app.module.ts to declare a new TodosComponent you can use inside your application.

1. If you want to try using your new component, check the selector name for your component inside todos.component.ts (Usually it will be something like app-todos or app-<component name>) and go to your app.component.html. Remove the contents inside it and type in your component selector name like this:

**<app-todos></app-todos>**

Re open your website and it should show what’s inside your todos.component.html which is a text:

todos works!

1. Now, we are going to use Bootstrap as our CSS frameworks, this will help us to establish a layout for our website as well as allowing us to use a prebuilt components inside our website.

A CSS framework is a library allowing for easier, more standards-compliant web design using the Cascading Style Sheets language. Most of these frameworks contain at least a grid.

You can read more about Bootstrap and its usage in here: <https://getbootstrap.com/>

You can play around and introduce more components or adjust the layout of your website in the future.

1. Bootstrap CDN

If you don't want to download and host Bootstrap yourself, you can include it from a CDN (Content Delivery Network). MaxCDN provides CDN support for Bootstrap's CSS and JavaScript.

You must also include jQuery by copying this chunk of code into your index.html inside the src folder:

<!-- Latest compiled and minified CSS -->  
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">  
  
<!-- jQuery library -->  
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>  
  
<!-- Latest compiled JavaScript -->  
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>

A screenshot of a computer

Description automatically generated

1. After that, go to src/app/app.component.html and open the app.component.html file, and remove everything inside (Ctrl+A / Command+A > Delete) and copy in this chunk of code.

This is the navigation bar I copied from the Bootstrap starter code with our todos component put below the navigation bar. Because in the todos component we have already defined a selector, we can call our component inside just by referencing it with **<app-todos></app-todos>**

<nav class="navbar navbar-inverse">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

<a class="navbar-brand" href="#">To-do List</a>

</div>

<div id="navbar" class="collapse navbar-collapse">

<ul class="nav navbar-nav">

</ul>

</div>

</div>

</nav>

<div class="container">

<div class="row">

<div class="col-md-8 col-md-offset-2">

<app-todos></app-todos>

</div>

</div>

</div>

1. Once you followed all the steps, your website should look like this, notice the navigation bar above the website styled by the Bootstrap and the todos component we just created below it. It only has todos works! text for now but we will make a to do list component soon to fill in that space.

Graphical user interface, application, Word

Description automatically generated

## Step 3: Building a To-do List Component

1. Go back to your todos component and go to your todos.component.html.

We will start by using this todos component template I created and start building it from scratch to cater to our needs:

**todos.component.html**

<div class="add-todo-form text-center">

<h1>Add Todo</h1>

<div class="form-group">

<input class="form-control input-lg" type="text" placeholder="Add Todo..." autofocus #todoText>

<br>

<button class="btn btn-primary btn-block">Create</button>

</div>

</div>

<div class="todo-list">

<div class="col-md-1">

<input type="checkbox">

</div>

<div class="col-md-9 pull-left">

Some Todo

</div>

<div class="col btns">

<input type="button" class="btn btn-default pull-left" value="Edit">

<input type="button" class="btn btn-danger pull-right" value="Delete">

</div>

</div>

There will be some classes and syntax that you are not familiar with, but don’t worry about that just yet. It is all just formatting and styling from the Bootstrap that I used.

In short, I added several texts, input text field and several buttons and checkboxes that we are going to use as we build our website.

1. Once you copied in the code and save it, your website should look like this now:

Graphical user interface, application

Description automatically generated

1. Now we have the user interface for our to do list, but it is not doing anything and is only filled with mock hard-coded data. Let’s use the back-end service we created yesterday to populate our data and give some life to the buttons so that it can function properly.

## Step 4: Setting Up the Backend Services

1. Now that we have the user interface for our to do list, we want to use our backend services and the data that we have in our database to show in our website.
2. Usually, the backend services will be readily available to use once you deploy it to your cloud platform and you will only need the endpoint URL to use it. But because we haven’t got to that (we will get to it in tomorrow class about deployment to cloud platform) we will serve it locally in order to be able to use it.
3. To do that, first we need to re-open the repository for our backend services that you created yesterday. Open a new window in your Visual Studio Code, click open folder and open the folder that you created in yesterday class (Back-end Building Basics). Once opened, open a new terminal and type in:

**npm start**

This command will serve you backend services in your local host, since the default port is 3000 for npm start you shouldn’t have any problem because we are using another port (4200) to serve our front-end code.

If there is no problem, you will see something like this in the terminal:

**Listening at http://localhost:3000**

1. Now that we have already a local server with our backend services, we need to set up our environment in for our website to call the backend services from our local host. Go back to your visual studio code for our front end side, and go to the environments folder. In the environments folder go to environment.ts and under the production:false line type in:

**baseURL: "http://localhost:3000/api/v1/bootcamp”**

export const environment = {

production: false,

baseURL: 'http://localhost:3000/api/v1/bootcamp'

};

This will set your base URL to endpoint calls to this address which is what we want.

1. Now that we have it all set up, let’s try calling one of the endpoint. We can first try to get our to do list.

To do that, go to your file explorer and right click on the app folder, create a new folder called services and right click on it to make a new file called todo.service.ts.

A screenshot of a computer

Description automatically generated with medium confidence

Copy in this chunk of starter code to start:

import { Injectable } from '@angular/core';

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

import { Observable } from 'rxjs';

import { environment } from '../../environments/environment';

@Injectable()

export class TodoService {

constructor(private *http*: HttpClient) { }

}

We are going to import some decorators and instance that will help us to integrate our backend services, as well as the environment where we are saving our base URL to our endpoint calls.

1. Now under the constructor, we will make a new function called getTodos(), this function will call our backend services and return the list of our to do list that we store in our database.

These lines of code will serve the purpose:

getTodos(): Observable<any> {

const url = environment.baseURL;

return this.http.get(url);

}

To explain it shortly, we will set our url to our baseURL that we defined in our environment file, and we will perform a http request using the imported httpClient to do that. Save the file.

1. After that we will have to import HttpClientModule into our app.module.ts as well to use it:

Go to your **app.module.ts** and import HttpClientModule by adding this at the top:

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

and also adding it into our list of imports inside our ngModule:

imports: [

BrowserModule,

HttpClientModule

],

Save the file.

1. Other than that, we also need to set our TodoService as one of our provider in our **app.component.ts**

Go to your **app.component.ts** and import your TodoService:

import { TodoService } from './services/todo.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.scss'],

providers: [TodoService]

})

Add your TodoService as providers inside the component definition.

1. Go back to your todos component and go to your todos.component.ts, we will be using the service to retrieve our to do list and show it in our website.
2. In our todos.component.ts, we will need to declare a variable that will store our to do list. To do that, under the class definition declare a variable called todos:

export class TodosComponent implements OnInit {

todos: Todo[] = [];

This will make a new variable called todos with an array of Todo datatype, we don’t have a Todo data type defined just yet, so we need to define it first by creating a new file under app folder.

**todo.ts**

export interface Todo{

text: string;

isCompleted: boolean;

\_id: string;

}

This will create a new interface called Todo in which inside it we have a text, isCompleted Boolean, and the id for the Todo item. Save the file.

1. Re-open your todos.component.ts and import both the TodoService and our Todo interface.

import { TodoService } from '../../services/todo.service';

import { Todo } from '../../todo';

Now that we have our TodoService imported, in our constructor we will need to make a new instance of todoService to use the service:

constructor(

private *todoService*: TodoService

) {}

1. For the endpoint call, we are going to call it inside our ngOnInit() so that it will be called everytime the page is initialized. Inside our ngOnInit called the getTodos() function that we defined in our TodoService.

ngOnInit() {

this.todoService.getTodos().subscribe({

next: (*todos*) => {

console.log(todos);

this.todos = todos;

},

error: (*error*) => {

console.log(error);

},

});

}

Subscribe() is a method in Angular that connects the observer to observable events. Whenever any change is made in these observable, a code is executed and observes the results or changes using the subscribe method. Subscribe() is a method from the rxjs library, used internally by Angular.

Once the endpoint call is finished it will return an array of todo objects that we have in our database if there is no error to the endpoint calls. We will then pass it on our todos object. I added in a console.log(todos) so that we can see what the endpoint has returned in our log later on. Save the file and go back to our website.

Now if you go to developer tools (Command + Option + I in Mac & Ctrl + Shift + I in Windows) and open the console you will see something like this:

Graphical user interface, application

Description automatically generated

If you open it, you can see what’s the response from our backend service because of the console logging we added earlier. If it has nothing don’t worry about it, that just means there is nothing in your database and we can create a new todo entry later on. If you open the network tab, and click on the bootcamp under the name tab you can also see the details of your endpoint calls.

Graphical user interface, text, application, email

Description automatically generated

## Step 5: Showing the Data Retrieved on the Website

1. Now that we have stored the to-do list we retrieved from our backend service into our to-dos variable, we would want to show it on our website. Go back to your **todos.component.html.**
2. Inside the todo-list div add in this code:

<div class="todo-list">

<div \*ngFor="let todo of todos; let i = index">

And add in another **</div>** to close it just before the last </div> so that it will look like this:

<div class="todo-list">

<div \*ngFor="let todo of todos; let i = index">

<div class="col-md-1">

<input type="checkbox">

</div>

<div class="col-md-9 pull-left">

Some Todo

</div>

<div class="col btns">

<input type="button" class="btn btn-default pull-left" value="Edit">

<input type="button" class="btn btn-danger pull-right" value="Delete">

</div>

</div>

</div>

\*ngFor is just another Angular directive that will let us loop our list of todos until it is exhausted. If you save the file, and check your website it will look like this if you have something in your to-do list inside the database.

In my case it has 4 entries so it will loop 4 times and create 4 to-do list items:

Graphical user interface

Description automatically generated

1. Go back to your **todos.component.html**, now we have 4 items but the name is still hard-coded and isn’t what we have in our database. To change the name erase the Some Todo text and replace it with:

{{todo.text}}

This is the Angular syntax for text Interpolation. Text interpolation lets you incorporate dynamic string values into your HTML templates. Use interpolation to dynamically change what appears in an application view, such as displaying the text that we have in our todos object. Save the file and go back to your website to see the changes. It will show the correct text now inside our to-do list.

Graphical user interface

Description automatically generated

1. Now the text has been changed, what about the check-list? Don’t worry about it, we are going to make use the isCompleted variable that we have in our object to handle that.

Go back to your todos.component.html and add checked property into your checkbox input:

[checked]="todo.isCompleted"

This is the full line for that (line 14):

<input type="checkbox" [checked]="todo.isCompleted">

This property will set the checked value for each checkbox based on the Boolean value inside our to-do list in our isCompleted variable. Save it and open your website, it will show the correct value now.

Graphical user interface

Description automatically generated

Congratulations, you have now successfully integrated your first backend services. But it doesn’t end here, now we can only view what’s inside our to-do list. But we would want to create a new to-do list, edit it and even delete it. So let’s do it and let’s integrate more services so that we can have more functionality.

## Step 6: Adding More Functionalities to our To-do List

1. First, we want to be able to create a new to-do list item. To do that, just like before we will have to create a new function in our service that can help us post a new to-do list item.

Go to your **todo.service.ts** and add this function:

**todo.service.ts**

postTodos(*param*: any): Observable<any> {

const url = environment.baseURL;

return this.http.post(url, param);

}

We are calling http.post because we want to add in a new entry into our database.

1. Go back to your **todos.component.ts**, and make a new variable called name under our todos variable:

This variable will be used to store whatever name is being typed in our add to-do text input.

todos: Todo[] = [];

name: any = '';

1. Create a new function called postTodos() underneath the closing bracket for our ngOnInit function:

postTodos() {

if (this.name === '') {

return;

} else {

const content = {

isCompleted: false,

text: this.name,

};

this.todoService.postTodos(content).subscribe({

next: (*response*) => {

location.reload();

console.log(response);

},

error: (*error*) => {

console.log(error);

},

});

}

}

This function will return/stop if the name is empty and it will make a new variable called content with isCompleted Boolean set to false as default and a text string that we are going to set to our name variable. It will then call the postTodos function that we just made if the name is defined. Once the function call is finished, it will refresh the client browser using location.reload() to show the newly posted to-do list item.

1. Go back to your **todos.component.html** to bind the name variable and the postTodos() function that we just created. To do that find the input text and add in [(ngModel)]=”name”. It should look like this:

<input class="form-control input-lg" type="text" [(ngModel)]="name" placeholder="Add Todo..." autofocus #todoText>

1. [(ngModel)] is just another Angular directive for two-way data binding that we can use to control a form. But first in order to be able to use it, we will need to import FormsModule into our **app.module.ts.**

**app.module.ts**

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

imports: [

BrowserModule,

HttpClientModule,

FormsModule

],

1. Now, we need to bind our function to our Create button inside our **todos.component.html**

<button class="btn btn-primary btn-block" (click)="postTodos()">Create</button>

The (click) properties will let us bind our postTodos() function to our button so that when we click it, it will initiate a function call to our postTodos().

1. Go ahead to your website and try it, type in a new to-do list item inside the Add Todo text field and then click create, it will call the backend service to post and create a new to-do list item for us.

Graphical user interface

Description automatically generated

1. Now, if we can add, we also want to delete our to-do list entries. To do that, we need to create another function called deleteTodo in our todo.service.ts. It will look like this:

**todo.service.ts**

deleteTodo(*id*: any): Observable<any> {

const url = environment.baseURL + '/' + id;

return this.http.delete(url);

}

The function will take the id for the entry and called http.delete to delete the entry from our database.

Note that we are adding + ‘/’ + id into our baseURL to pass a parameter to our backend services call because we need the corresponding id to delete the right entry.

1. Go back to your todos.component.ts and create a new function called deleteTodo.

**todos.component.ts**

deleteTodo(*i*: any) {

const id = this.todos[i].\_id;

this.todoService.deleteTodo(id).subscribe({

next: (*response*) => {

console.log(response);

location.reload();

},

error: (*error*) => {

console.log(error);

},

});

}

This function will take an index as parameter and retrieve the id from the corresponding todos entry. After that, it will call the deleteTodo function that we just defined to delete the entry from our database, it will then refresh the browser to show the changes.

1. Now that we have created the function, we need to once again bind it to our buttons, go back to your todos.component.html and find your delete button and bind the function to it with (click) once again:

**todos.component.html**

<input type="button" class="btn btn-danger pull-right" (click)="deleteTodo(i)" value="Delete">

1. Now go back to your website and press delete on any of your entry, it will remove it from the to-do list by calling the delete endpoint from our backend services and refresh the page to reflect the changes.
2. Now that we have create and delete, the last thing we need will be update. We are going to make an update function and we are going to use it for two cases:
   1. User click on the checkboxes to change the completion status of his/her to-do list entry
   2. User click on edit to change the name of his/her to-do list entry
3. To do that, we will first create a new function in our **todo.service.ts** called updateTodo.

**Todo.service.ts**

updateTodo(*param*: any, *id*: any): Observable<any> {

const url = environment.baseURL + '/' + id;

return this.http.put(url, param);

}

This function will take 2 parameters, the first one is the to-do-list entry filled with name and isCompleted Boolean (Update Object) and the id of the object that we want to update.

1. Go to your **todos.component.ts** and create a new function called checkTodo:

**todos.component.ts**

checkTodo(*i*: any) {

const id = this.todos[i].\_id;

const content = {

isCompleted: !this.todos[i].isCompleted,

text: this.todos[i].text,

};

this.todoService.updateTodo(content, id).subscribe({

next: (*response*) => {

location.reload();

console.log(response);

},

error: (*error*) => {

console.log(error);

},

});

}

This function will take an index and get the corresponding id of the to-do list entry that we want to update, it will then update the isCompleted Boolean to the complement of its original value and send it as parameter to our updateTodo function. Once finished it will refresh the page to update the value to the correct value. This function will be used for our checkboxes function.

1. Bind it to your checkboxes by using (click) property like before:

<input type="checkbox" [checked]="todo.isCompleted" (click)="checkTodo(i)">

Go back to your website and test it, the checkbox button should work now. You can update the isCompleted flag of your to-do list entry from there now.

1. Now, we want to allow the user to edit the name of the to-do list entry as well. We can use the same backend service that we created to do that. But we need to make a new function called editTodo in our **todos.component.ts.**

This one is a bit complex because we want to have two modes. One is view only mode and one is editing mode, in order to do that we need to introduce an array of flag and save it into a variable called editMode. This flag will determine if the entry is in edit mode or view only mode.

**todos.component.ts**

export class TodosComponent implements OnInit {

todos: Todo[] = [];

name: any = '';

editMode: any[] = [];

Other than that we also need to initialize the value for our editMode, go to our ngOnInit function and change the getTodos function call, the new function call will include a loop that initialize the values of our editMode array, it is set to false as default.

this.todoService.getTodos().subscribe({

next: (*todos*) => {

console.log(todos);

this.todos = todos;

for (let i = 0; i < todos.length; i++) {

this.editMode.push(false);

}

},

error: (*error*) => {

console.log(error);

},

});

Now add in the editTodo function below our checkTodo function:

editTodo(*i*: any) {

if (this.editMode[i]) {

const id = this.todos[i].\_id;

const content = {

isCompleted: this.todos[i].isCompleted,

text: this.todos[i].text,

};

this.todoService.updateTodo(content, id).subscribe({

next: (*response*) => {

location.reload();

console.log(response);

},

error: (*error*) => {

console.log(error);

},

});

} else {

this.editMode[i] = true;

}

}

This function will do two things:

1. If we are already in editMode, it will update the corresponding to-do list entry to the text we typed in when we are editing the entry and refresh the page.
2. If not, it will only set the editMode flag for that entry to true and allow us to type the name we want to change it into.
3. In order to achieve that, we will also need to set the conditions in our **todos.component.html** as well.

Change {{todo.text}} into these lines:

**todos.component.html**

<div \*ngIf="!editMode[i]"> {{todo.text}} </div>

<div \*ngIf="editMode[i]">

<input class="form-control" type="text" [(ngModel)]="todo.text" autofocus #todoText>

</div>

It should look like this now inside the div:

Graphical user interface, text

Description automatically generated

1. Other than that, we also need to bind our editTodo function to our edit button, this will do the job:

<input type="button" class="btn btn-default pull-left" (click)="editTodo(i)" value="Edit">

1. Now, if you do everything correctly, and saves all the page. Once you go to your website, the edit function should work correctly. On your first click, it will show a text input where you can adjust your name to another name. After you finish, you can click on the edit button again and it will call the updateTodos service to update the corresponding to-do list entry and change its name. It will look like this:

Graphical user interface

Description automatically generated

1. Congratulations, you have now built a working To-do List website with all the correct functionalities.

We will now talk a little bit about testing / unit test process.

## 

## Step 7: Testing and Unit Tests

1. Go to your **todos.component.spec.ts** and import your TodoService and some modules needed for testing.

import { TodoService } from '../../services/todo.service';

import { HttpClientTestingModule } from '@angular/common/http/testing';

import { waitForAsync } from '@angular/core/testing';

1. In your configureTestingModule import the HttpClientTestingModule as well TodoService as our providers.  
   It should look like this:

beforeEach(async () => {

await TestBed.configureTestingModule({

imports: [ HttpClientTestingModule ],

declarations: [ TodosComponent ],

providers: [ TodoService ]

})

.compileComponents();

});

1. First of all, we are going to test if the array name is empty on initialization, add this under the first test:

it('name should be empty string', () => {

expect(component.name).toBe('');

});

1. After that, let’s check if our todos array and editMode flag array is empty on initialization:

To do that, add this under the second test, it will check if our arrays length are 0 and its size is 0:

it('should be empty array', () => {

expect(component.todos.length).toEqual(0);

expect(component.todos.length).toHaveSize(0);

expect(component.editMode.length).toEqual(0);

expect(component.editMode.length).toHaveSize(0);

});

1. Last but not least, let’s check if our service is working properly, to do that add in this example test for checking if our getTodos are returning the correct result:

it('should return data from service function getTodos', waitForAsync(() => {

const result = [] as any;

let mockTodoService: jasmine.SpyObj<TodoService>;

mockTodoService = jasmine.createSpyObj('TodoService', ['getTodos']);

mockTodoService.getTodos.and.returnValue(result);

fixture.detectChanges();

expect(component.todos).toEqual(result);

}))

1. Once you are done, go to your **app.component.spec.ts** and remove these two tests because we are not using it anymore in our tests (this tests are being generated automatically during the project creation).

Text

Description automatically generated

1. Open up a new terminal and type in this command

**npm run test**

It will open up a new window and automatically launch the unit tests you just made, see if you pass all the tests, if you didn’t then there might be something wrong with your component.

A picture containing graphical user interface

Description automatically generated

Congratulations, you have just made your first set of unit tests.

Can you think of more test cases?

Try to write one of your own and we will talk about it more tomorrow.