**Creating a New Project**

In this lesson, we'll get started with the next project that we'll be tackling in the next couple of sections.

**We'll cover the following**

* [New project](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5430415109652480#New-project)
* [Installing Bootstrap](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5430415109652480#Installing-Bootstrap)
* [Project architecture](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5430415109652480#Project-architecture)

The next project we’ll be working on will be an application in which visitors can search for cocktails. This will require an extensive database of alcoholic beverages. Fortunately, there are APIs that we can use to query for a list of cocktails. For this project, we’ll be using a service called the Cocktail DB. It’s completely free.

You can find more information about its API [here](https://www.thecocktaildb.com/api.php?ref=apilist.fun).

We’re going to be focusing on multiple topics for this project. The main objectives of this project are to learn about the following:

* Services
* Performing HTTP requests
* Security
* RxJS

**New project**

If you’re running code locally, then you’ll need to run the following command:

ng new cocktails

During the setup process, the CLI will ask you to configure the project. The default settings work fine. We won’t need routing for this project.

**Installing Bootstrap**

After installing the project, navigate to the newly created directory and install Bootstrap. We’ll be using it to help us with styles. You can run the following command to install Bootstrap:

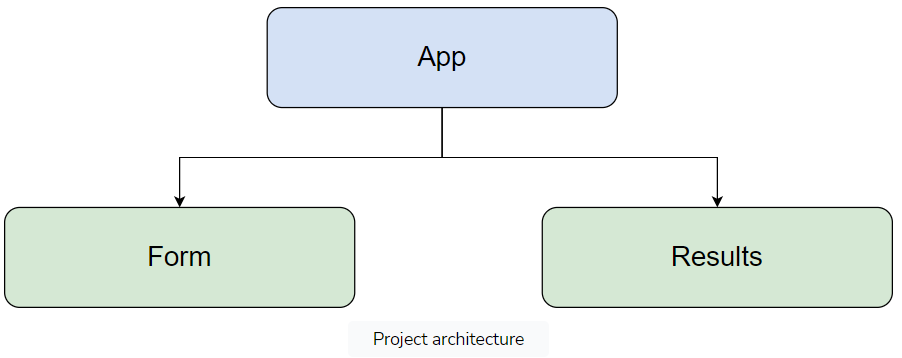
npm i bootstrap

We’ll need to update the styles.css file to import Bootstrap.

@import "bootstrap/dist/css/bootstrap.css"

**Project architecture**

The overall structure of the project will be simple. We’ll have a form to perform a search and a table to display the list of results. It would be a good idea to put these two sections into their own components to keep things organized. Here’s what the project architecture will look like:



Let’s create the components that will house the form and results. In the command line, run the following command to generate the component for the form:

ng generate component form

Then this command for the results:

ng generate component results

Next, we’ll add some HTML to the component’s templates to make things look appealing. In the form.component.html template file, we’ll add the following:

<div·class="card·mt-3">

··<div·class="card-body">

····<form·class="input-group·mb-3">

······<input·type="text"·class="form-control">

······<div·class="input-group-append">

········<button·class="btn·btn-primary"·type="submit">Search</button>

······</div>

····</form>

··</div>

</div>

We aren’t going to use any directives or bindings yet. We’re just trying to get the visual aspect out of the way with Bootstrap’s classes.

Next, we’ll update the results.component.html template file.

<table·class="table">

··<thead>

····<tr>

······<th>Drink</th>

······<th>Instructions</th>

····</tr>

··</thead>

··<tbody>

····<tr>

······<td>Vodka</td>

······<td>Here's·some·instructions.</td>

····</tr>

··</tbody>

</table>

Lastly, we’ll load both components in the app.component.html template file.

<div·class="container">

··<app-form></app-form>

··<app-results></app-results>

</div>

We don’t need to import or register the components. Angular has already done so for us when we generated the components.

**Submitting the Form**

In this lesson, we'll focus on submitting the form.

**We'll cover the following**

* [Storing the query](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5642709303296000#Storing-the-query)
* [Handling form submission](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5642709303296000#Handling-form-submission)
* [Child to parent communication](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5642709303296000#Child-to-parent-communication)

The first thing we’ll handle is the form submission. The Cocktail DB API requires that we provide it with a search query to filter the results of beverages. We have a form that will allow the user to input a search query. We’ll need to store the query and then submit it when the form is submitted.

**Storing the query**

We’ll create a property, called query, in the form.component.ts class file.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

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

@Component({

  selector: 'app-form',

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

  styleUrls: ['./form.component.css']

})

export class FormComponent implements OnInit {

  query = '';

  constructor() { }

  ngOnInit(): void {

  }

}

Next, we’ll update the query property whenever the user types in the form. Alternatively, we can retrieve the value the minute the user submits the form. However, we want to keep track of the latest value to be aware of the latest updates in the DOM. This is beneficial because we’ll have the value at hand whenever we need it.

In the form.component.html template file, we’ll update the <input> element to listen for the input event.

1

2

3

4

5

6

7

8

9

10

<div class="card mt-3">

  <div class="card-body">

    <form class="input-group">

      <input type="text" class="form-control" (input)="query = $event.target.value">

      <div class="input-group-append">

        <button class="btn btn-primary" type="submit">Button</button>

      </div>

    </form>

  </div>

</div>

If the event is triggered, we’ll update the query property with the $event.target.value property.

**Handling form submission**

Our next step in this journey is to listen to when the form is submitted. We’ll add an event listener to the <form> element in the form.component.html template file.

<div class="card mt-3">

  <div class="card-body">

    <form class="input-group" (submit)="onSubmit($event)">

      <input type="text" class="form-control" (input)="query = $event.target.value">

      <div class="input-group-append">

        <button class="btn btn-primary" type="submit">Button</button>

      </div>

    </form>

  </div>

</div>

We’re listening to the submit event. If triggered, we’ll call a function, called onSubmit(), where we’ll provide it with the $event object. We’re providing it this object because we don’t want the page to refresh. We’ll stop the default behavior in the function.

Let’s define it in the component class file.

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

@Component({

  selector: 'app-form',

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

  styleUrls: ['./form.component.css']

})

export class FormComponent implements OnInit {

  query = '';

  constructor() { }

  ngOnInit(): void {

  }

  onSubmit(event: Event) {

    event.stopPropagation();

  }

}

Inside the function, we’re calling the preventDefault() function to stop the event from refreshing the page.

**Child to parent communication**

The last thing we’ll do is tell the app component to perform the search. We’re performing the search in the app component because that we’ll need to provide the results component with the results. It’s much easier to communicate the results from parent to child instead of sibling to sibling.

In the form.component.ts class file, we’ll update the onSubmit() function to let the parent component know the form was submitted.

import { Component, OnInit, Output, EventEmitter } from '@angular/core';

@Component({

  selector: 'app-form',

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

  styleUrls: ['./form.component.css']

})

export class FormComponent implements OnInit {

  @Output() submitted = new EventEmitter();

  query = '';

  constructor() { }

  ngOnInit(): void {

  }

  onSubmit(event: Event) {

    event.preventDefault();

    this.submitted.emit(this.query);

  }

}

In the code snippet above, we’re importing the Output and EventEmitter classes. They’ll help us communicate with the parent component. We’re creating a property called submitted, which will be set to a new instance of the EventEmitter class. We’ll need it to emit an event that the parent component can listen to.

In the onSubmit() function, we’re calling the emit() function with the query property.

Let’s listen for the submitted event in the app.component.html template file.

<div class="container">

  <app-form (submitted)="search($event)"></app-form>

  <app-results></app-results>

</div>

If the event is triggered, we’ll call a function called search(). We’re providing the search() function with the $event object. However, keep in mind that this is a custom event. The $event object we’re providing is not the typical event object in JavaScript. It represents whatever value we emitted from the event. In our case, this would be the search query.

The last thing to do is to define the search() function in the app.component.ts file.

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  search(query: string) {

    console.log(query);

  }

}

We’re logging the query. Here’s the updated code. We’ll see the query in the console if we submit the form.

**Generating Services**

In this lesson, we'll learn what services are and how to create them.

**We'll cover the following**

* [What are services?](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4642242976808960#What-are-services)
* [Generating a service](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4642242976808960#Generating-a-service)
* [Using the service](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4642242976808960#Using-the-service)

Our form is ready. It’s sending the search query to the App component, where we’ll perform the request for the data. There are two approaches we can take for making a request. The first is to request the data in the component. The second is to request the data in a **service**.

**What are services?**

Services are objects with methods and properties for fetching data. They’re responsible for creating, updating, and fetching data. We don’t have to use services. They’re a completely optional feature of Angular.

If we want to, we can request the data from the component. However, chances are we’ll want to make multiple requests to the same data source. This means we’ll have to retype the request in every component in which we’d like to perform a request.

By outsourcing requests to an object, we only need to type the request once. Then, we can import the service into **any** component and use the service to make the request for us. The goal of a service is to **supply data to a component(s)**. Anytime you need to make an API request, you’ll want to put it in a service.

**Generating a service**

Services are created with the ng generate service <name> command. What a surprise, right? We’ll be generating a service to handle requests to the Cocktail DB API.

Let’s run the following command in our project:

ng generate service cocktail

This will create two files: cocktail.service.ts and cocktail.service.spec.ts. The spec file in its name is the testing file for the service. Testing is something we’ll look at in another section.

Our main focus will be on the cocktail.service.ts file.

**Using the service**

One thing you may notice during the generation of the service file is that the CLI did not register the service anywhere. This is because services are not bound to any component or module. They can be used anywhere.

If we want to use the service, we’ll need to import it. We’ll be using the service in the App component. We’ll update the app.component.ts to the following:

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

import { CocktailService } from './cocktail.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  constructor(private cocktail: CocktailService) { }

  search(query: string) {

    console.log(query);

  }

}

Two things are going on here. We’re importing the CocktailService object. Next, we’re adding a constructor() function in which we’re defining a new property called cocktail. We’ve annotated it with the CocktailService object.

The service is now accessible in the component. This is very odd. How are we able to use the service when we didn’t create an instance of it? Angular does something behind the scenes called **dependency injection**. We’ll look at what that is in the next lesson to understand what’s going on.

**Understanding Dependency Injection**

In this lesson, we'll take the time to learn about dependency injection in Angular.

**We'll cover the following**

* [Drawbacks without dependency injection](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4582353684398080#Drawbacks-without-dependency-injection)
* [Dependency injection design pattern](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4582353684398080#Dependency-injection-design-pattern)
* [Dependency injection in Angular](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4582353684398080#Dependency-injection-in-Angular)
* [Using a dependency](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/4582353684398080#Using-a-dependency)

Dependency injection is one of those things that can be difficult to wrap your head around. It’s a feature in Angular that makes our lives so much easier. Let’s look at why dependency injection is great and how it works in Angular.

**Drawbacks without dependency injection**

We’ll step away from our project for a moment. First, we’re going to look at some of the problems we’ll encounter without dependency injection.

Let’s look at an example.

class engine {

}

class car {

  engine;

  constructor() {

    this.engine = new engine();

  }

}

In the example above, we have two classes: engine and car. The car class creates a new instance of the engine object. It’s assigned to a property called engine. This makes the engine class a *dependency* of the car class. It relies on the engine class to work.

This example works, but it isn’t scalable. Let’s see what would happen if we added a property to the engine class.

class engine {

  fuel;

  constructor(fuel){

    this.fuel = fuel;

  }

}

class car {

  engine;

  constructor() {

    this.engine = new engine('gas');

  }

}

This time around, we’re creating a property, called fuel, in the engine class. Not only did we update the engine class, but we had to update the car class. This is a problem because we have to continually go back and forth between files whenever a dependency changes. At the moment, it’s manageable because we have two classes. It’s possible for our dependency to have its own dependency. What if our car class has more dependencies? It can easily become a disaster to manage.

**Dependency injection design pattern**

Dependency injection is a design pattern for making classes reusable and testable. Let’s look at the same example with dependency injection.

class car {

  engine;

  constructor(engine) {

    this.engine = engine;

  }

}

In this example, we’re expecting an object called engine. We aren’t creating an instance of the engine object. It’s made for us and is supplied when the car class is instantiated.

This is dependency injection. The responsibility of creating the dependency is no longer in our hands. Instead, it’s created beforehand. If the engine class changes, we don’t have to update the car class anymore.

Here’s how an instance of the car class can be created:

const newEngine = new engine('gas');

const newCar = new car(newEngine);

With all that being said, dependency injection isn’t all that perfect. A problem that arises with dependency injection is having to keep track of the dependency structure. Dependencies can have their own dependencies. As we add more dependencies, we have more to manage.

**Dependency injection in Angular**

Angular heavily utilizes dependency injection. It’s able to solve the issue of having to keep track of dependencies. Angular will manage the dependencies for you. Here’s how dependency injection works in Angular:

1. Define a class to be used as a dependency. Most of the time, it’s a service.
2. Register it with Angular.
3. Declare it as a dependency in a component.

Let’s look at the first two steps by opening the cocktail.service.ts class.

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

@Injectable({

  providedIn: 'root'

})

export class CocktailService {

  constructor() { }

}

Here, we have a class decorated with the Injectable decorator. This will tell Angular to add this class to the list of injectable classes. It will take care of creating a new instance of the class for you. This is why we don’t see new CocktailService() anywhere.

In the @Injectable() decorator, we’re telling Angular where to expose the class by setting the providedIn option. The value for this must be where we’d like to use the class. By default, the CLI will set this to root. If set to root, Angular will provide it to all components registered under the root module of the application. In our case, this would be the app module.

One important thing to understand about dependency injection is that Angular will create *one* instance of the class. If we use the service in multiple classes, then those classes will use the same instance. They don’t get their own instance.

**Using a dependency**

Let’s go back to the app.component.ts file. We have one line that’s pretty magical.

constructor(private cocktail: CocktailService) { }

During the compilation process, Angular scans our components for the dependencies it needs. Specifically, it will look at a component’s constructor() function. It will notice that we want an instance of the CocktailService because we annotated the parameter with it. It will then take care of injecting the dependency for you. This is why we never have to do something like this in our class.

export class AppComponent {

  cocktail;

  constructor(CocktailService) {

    this.cocktail = new CocktailService();

  }

}

Angular will create an instance for us behind the scenes making our job so much easier.

Dependency injection is also available for pipes and directives. For example, if you can recall, we created a class directive for dynamically adding classes to an element. The constructor() function looked something like this:

constructor(private el: ElementRef) { }

We weren’t creating an instance of the ElementRef class. Angular saw that we wanted an instance of the ElementRef class and took care of providing one for us that was automatically connected to the element on which we had the directive.

Dependency injection in Angular helps us by providing us with instances of classes that we’ll need to make the app functional. We never have to worry about setting up the dependencies. They’re ready out of the box.

**HTTP Requests**

In this lesson, we'll learn how to make an HTTP request with Angular.

**We'll cover the following**

* [The HttpClientModule](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6475228047212544#The-HttpClientModule)
* [Making a request](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6475228047212544#Making-a-request)
* [Retrieving the results](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6475228047212544#Retrieving-the-results)

It’s time to make a request to the Cocktail DB API. If you’ve developed some applications previous to this course, you may already be familiar with how requests work. They can be made with the XMLHttpRequest object or the fetch API. There are also libraries, such as Axios, that simplify making a request.

Angular is a framework that comes with a solution baked in. It has an HTTP client we can use to perform requests. We won’t need to use 3rd party libraries to help us with this.

**The HttpClientModule**

By default, the HTTP client is not available in our project. We need to import a module that will expose the HTTP client to our application. We’ll be able to make requests anywhere in our application after we’ve imported it.

In the app.module.ts file, we’ll update the class file to use the HttpClientModule.

import { BrowserModule } from '@angular/platform-browser';

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

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

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

import { FormComponent } from './form/form.component';

import { ResultsComponent } from './results/results.component';

@NgModule({

  declarations: [

    AppComponent,

    FormComponent,

    ResultsComponent

  ],

  imports: [

    BrowserModule,

    HttpClientModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

The HttpClientModule can be imported via the @angular/common/http package.

**Making a request**

Our next step is to make a request with Angular. In the cocktail.service.ts file, we’ll grab the HttpClient class as a dependency.

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

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

@Injectable({

  providedIn: 'root'

})

export class CocktailService {

  constructor(private http: HttpClient) { }

}

The HttpClient class comes with methods for performing requests. We’ll need it if we want to communicate with the Cocktail DB API. We require it as a dependency by adding it to the constructor() function’s parameters. Angular will inject it during compilation.

Next, we’ll create a method, called search(), in the service.

export class CocktailService {

  constructor(private http: HttpClient) { }

  public search(query: string) {

    return this.http.get('https://www.thecocktaildb.com/api/json/v1/1/search.php', {

      params: {

        s: query,

      }

    });

  }

}

The search() method has one parameter called query. This will be the search term the user can input to search for specific beverages.

Inside the function, we’re calling the this.http.get() function. This will make a GET request. There are two arguments. The first is the URL. Information about the API URL can be found [here](https://www.thecocktaildb.com/api.php?ref=apilist.fun).

We can add query parameters to the URL to filter the results. We’ll want to take advantage of this. The second argument to the function is an object of settings for the request. We’re passing it in with a property called params. This is where we can add query parameters to the URL.

We’re adding one parameter called s. This is a parameter to which we can add the query.

This will return an **observable**. Observables are a topic we’ll explore in the next section. All you need to know is that this will return an object from which we can grab the data.

**Retrieving the results**

We’ll want to grab the results from the search() method. First, we have to call it from the app.component.ts class file. We’ll update it to the following:

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

import { CocktailService } from './cocktail.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  constructor(private cocktail: CocktailService) { }

  search(query: string) {

    this.cocktail.search(query).subscribe(response => {

      console.log(response);

    });

  }

}

In the component, we’re updating the search() function to call the search method on the CocktailService. Keep in mind that an observable is returned. This is an object with a method, called subscribe() that we can call to grab the list of results.

We’re passing an arrow function in which we’re provided an argument called response. This argument is logged. We’ll deal with displaying results in the next lesson. For now, we want to verify that the request is working.

**Displaying the Results**

In this lesson, we'll learn how to display the results to the user.

**We'll cover the following**

* [Storing the results](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5229806884487168#Storing-the-results)
* [Parent to child communication](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5229806884487168#Parent-to-child-communication)
* [Looping through the drinks](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5229806884487168#Looping-through-the-drinks)

We’ve got everything we need. The last step to building our application is to display the results. Here’s what we’ll need to do:

1. Store the results
2. Send the results down to the child component
3. Loop through the results

Before proceeding, try to display the results on your own. We’ve already covered pretty much everything you’ll need to know to accomplish this task. Good luck!

**Storing the results**

First, we’ll want to store the results. This way, we can send them down to the Results component. In the app.component.ts file, we’ll update the class to the following:

export class AppComponent {

  drinks = [];

  constructor(private cocktail: CocktailService) { }

  search(query: string) {

    this.cocktail.search(query).subscribe((response: any) => {

      this.drinks = response.drinks;

    });

  }

}

We’re creating a new property called drinks. It will be set to an empty array. We’re updating the drinks property after we’ve received a response from the request. The cocktail DB API will return an object with a property called drinks. Its value will be an array.

One thing worth pointing out is that we’ve annotated the response parameter to any. Otherwise, the compiler would have complained.

**Parent to child communication**

The next step is to send the drinks property to the results component. We’ll bind a property to the <app-results> component.

<div class="container">

  <app-form (submitted)="search($event)"></app-form>

  <app-results [drinks]="drinks"></app-results>

</div>

Next, we’ll update the results.component.ts file to accept the binding.

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

@Component({

  selector: 'app-results',

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

  styleUrls: ['./results.component.css']

})

export class ResultsComponent implements OnInit {

  @Input() drinks = [];

  constructor() { }

  ngOnInit(): void {

  }

}

We’re importing the Input decorator so we can tell Angular that we’re expecting a property called drinks. We’ll set the initial value to an array.

**Looping through the drinks**

The last step is to display the results. We’ll do that in the results.component.html template file.

<table class="table">

  <thead>

    <tr>

      <th>Drink</th>

      <th>Instructions</th>

    </tr>

  </thead>

  <tbody>

    <tr \*ngFor="let drink of drinks">

      <td>{{ drink.strDrink }}</td>

      <td>{{·drink.strInstructions·}}</td>

    </tr>

  </tbody>

</table>

In the template, we’re using the ngFor directive on the <tr> element to loop through the drinks array. We’re assigning each iteration in the loop the alias of drink. Then, we’re referencing the properties in their respective locations. We’re grabbing the name of the drink and instructions to create the drink.

The API returns an array of objects. Each object has two properties: strDrink and strInstruuctions. Feel free to look at the API for more information about the data you can expect to find from the response.

Below is the updated code. If we were to perform a search, we’d be able to view a list of results. We’ve successfully finished the app. In the next couple of lessons, we’ll add some finishing touches.

**Highlighting Keywords**

In this lesson, we'll look at how we can highlight keywords

**We'll cover the following**

* [Passing down the query](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6142065890557952#Passing-down-the-query)
* [Accepting the query](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6142065890557952#Accepting-the-query)
* [Adding HTML](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/6142065890557952#Adding-HTML)

One cool feature common in most search engines is highlighting the keyword in the search results. We can easily highlight keywords with Bootstrap. There’s a class called font-weight-bold, which will make a text look bold.

To accomplish this, we’ll need to provide the results component with the search term. This way, we’ll be able to know which keyword to highlight in the results.

**Passing down the query**

We’ll be doing the same thing we did before. We’ll store a copy of the search query in the AppComponent class. Then, we’ll send it down. In the app.component.ts file, we’ll add a property called query.

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

import { CocktailService } from './cocktail.service';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  drinks = [];

  query = '';

  constructor(private cocktail: CocktailService) { }

  search(query: string) {

    this.query = query;

    this.cocktail.search(query).subscribe((response: any) => {

      this.drinks = response.drinks;

    });

  }

}

In the example above, we added the query property with an initial value of an empty string. We update this property in the search() method. We’re setting the query before we query the API.

Next, we’ll update the template to bind a property to the <app-results> component.

<div class="container">

  <app-form (submitted)="search($event)"></app-form>

  <app-results [drinks]="drinks" [query]="query"></app-results>

</div>

**Accepting the query**

In the results.component.ts class component file, we’ll add a new property with the @Input() decorator. This will tell Angular to expect a property from the parent component.

export class ResultsComponent implements OnInit {

  @Input() drinks = [];

  @Input() query = '';

  constructor() { }

  ngOnInit(): void {

  }

}

**Adding HTML**

The next part will be a bit tough. With the query in hand, we can begin to highlight the keyword in the results. The question is: How? We can use regular expressions on the drink and instructions before outputting them on the template.

We’ll create a function that will perform the regular expression on the string.

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

@Component({

  selector: 'app-results',

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

  styleUrls: ['./results.component.css']

})

export class ResultsComponent implements OnInit {

  @Input() drinks = [];

  @Input() query = '';

  constructor() { }

  ngOnInit(): void {

  }

  boldStr(str: string) {

    if (this.query.length < 3) {

      return str;

    }

    const reg = new RegExp('(' + this.query + ')', 'gi');

    return str.replace(reg, '<span class="font-weight-bold">$1</span>');

  }

}

The function we defined is called boldStr. It has one parameter, which is the string to perform a search on for the keyword. Before we initiate a search, we’re making sure the query property is at least three characters long. This is to prevent from highlighting the entire string. If it is less than three characters, we’ll just return the original string.

Afterward, we’re creating a new instance of the RegExp object. The regular expression we’re creating will check if the string contains a query. We’re also adding case insensitive matches so that we don’t miss anything.

Lastly, we’re calling the replace() method on the string with the regular expression object. If we find a match, we’ll replace it with a <span> element wrapped around the query with the font-weight-bold class.

Let’s apply it to the results.component.html template.

<table·class="table">

··<thead>

····<tr>

······<th>Drink</th>

······<th>Instructions</th>

····</tr>

··</thead>

··<tbody>

····<tr·\*ngFor="let·drink·of·drinks">

······<td>{{·boldStr(drink.strDrink)·}}</td>

······<td>{{·boldStr(drink.strInstructions)·}}</td>

····</tr>

··</tbody>

</table>

Here’s the updated code:

**Understanding XSS Attacks**

In this lesson, we'll look at how XSS attacks work.

**We'll cover the following**

* [How to insert content](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5363124246413312#How-to-insert-content)
* [XSS attack example](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5363124246413312#XSS-attack-example)
* [How Angular handles XSS](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/5017901150502912/5363124246413312#How-Angular-handles-XSS)

In the previous lesson, we encountered an issue when trying to insert raw HTML into the document with interpolation. This is because Angular is attempting to prevent something called an XSS attack.

XSS is when malicious HTML is inserted into the document. This type of attack can be used to steal login credentials or sensitive data. It’s possible for the user to be redirected to a page they weren’t originally intended to be redirected to.

**How to insert content**

Let’s look at how we can insert content into a document.

**OutputJavaScript**

const h1 = '<h1>Hello!</h1>';

document.body.innerHTML = h1;

Save

Reset

In the example above, we’re creating an <h1> element. This is then inserted into the document by updating the document.body.innerHTML property. If we look at the output, we’ll be able to view the element on the document. It doesn’t have the surrounding <h1></h1> tags. The browser will process the content before it displays it on the page.

While convenient, it does leave us vulnerable to XSS attacks, also known as cross-site scripting.

**XSS attack example**

Let’s look at how an XSS attack can be performed.

**OutputJavaScript**

const xss = `<img src="" onerror="console.log('XSS')">`;

document.body.innerHTML = xss;

In this example, we’re trying to insert an image without a source. The browser will throw an error because it can’t find the image. We can catch the error by adding the onerror attribute. Its value can be some JavaScript we’d like to execute.

Let’s imagine we were trying to insert some HTML from a 3rd party source. If the source is compromised, they can potentially hijack the application by running JavaScript. If we were to view the output, we’d see the message logged by the onerror attribute. This is the kind of thing we want to avoid.

**How Angular handles XSS**

Angular will encode the characters in a string before it inserts it into the document. This process is known as escaping.

**OutputJavaScript**

let xss = `<img src="" onerror="console.log('XSS')">`;

xss = xss.replace(/</g, '&lt;').replace(/>/g, '&gt;');

document.body.innerHTML = xss;

In the example above, we’re calling the replace() method twice on the string to replace the < and > characters. We’re replacing them with their respective entities. For a list of entities, check out this [link](https://dev.w3.org/html5/html-author/charref).

This is more or less what’s going on internally in Angular. It’s escaping the strings before they get inserted into the document to prevent XSS attacks. You can think of it as purifying the HTML.

Angular does an excellent job of preventing XSS attacks. One good resource for learning about the various attacks can be found [here](https://owasp.org/www-community/xss-filter-evasion-cheatsheet).

I highly recommend checking it out to learn more about how XSS can be performed.

# Inserting Raw HTML

In this lesson, we'll learn how to insert raw HTML.

We saw how Angular takes care of preventing XSS attacks in the previous lesson. It’s something we generally don’t have to worry about. However, there are some cases where we’ll want to insert HTML into the document.

We can do so by binding the innerHTML property on the element. Let’s update the results.component.html template file to see what that looks like.

If we run the example, we’ll see the keywords highlighted in the document without seeing the elements in plain text. Surprisingly, Angular allows us to bind the properties of an element on the element. This allows us to insert raw HTML when we need to.

One thing to keep in mind is that Angular will continue to purify your HTML. It’s just that, this time, it won’t be as restrictive. For example, let’s say we had the following in the class:

xss = '<img src="" onerror="console.log(1)">';

We could bind this property to an element on the template.

<p [innerHTML]="xss"></p>

Angular will still escape the characters in this case. So, in a way, we get the best of both worlds. We can dynamically insert HTML into a document while still having Angular perform a security check on the string.