

MEAN STACK TECHNOLOGIES-MODULE II- ANGULAR JS, MONGODB (Skill Oriented Course)

S.NO	DATE	NAME OF THE EXPERIMENT	PAGE.NO	MARKS AWARDED
1.a		Angular Application Setup	1-2	
1.b		Components and Modules	2-5	
2.a		Structural Directives - ngIf	6-14	
2.b		ngFor	14-15	
3.a		Attribute Directives - ngStyle	16-18	
3.b		ngClass	18-20	
4.a		Property Binding	21-22	
4.b		Attribute Binding	22-23	
5.a		Built in Pipes	24-25	
5.b		Passing Parameters to Pipes	25-26	
6.a		Passing data from Container Component to Child Component	27-28	
6.b		Passing data from Child Component to ContainerComponent	28-31	
7.a		Template Driven Forms	32-33	
7.b		Model Driven Forms or Reactive Forms	33-35	
8.a		Custom Validators in Template Driven forms	36-39	
8.b		Services Basics	39-40	
9.a		Server Communication using HttpClient	41-42	
9.b		Communicating with different backend services using Angular HttpClient	42-43	
10.a		Routing Basics, Router Links	44-46	
10.b		Route Guards	46-48	
11.a		Installing MongoDB on the local computer, Create MongoDB Atlas Cluster	49-51	
12.a		Create and Delete Databases and Collections	52-54	
12.b		Introduction to MongoDB Queries	54-55	

1a). Course Name: Angular JS

Module Name: Angular Application Setup

Observe the link <http://localhost:4200/welcome> on which the mCart application is running. Perform the below activities to understand the features of the application.

To develop an application using Angular on a local system, you need to set up a development environment that includes the installation of:

- Node.js (^12.20.2 || ^14.15.5 || ^16.10.0) and npm (min version required 6.13.4)
- Angular CLI
- Visual Studio Code

Install Node.js and Visual Studio Code from their respective official websites.

Steps to install Angular CLI

Angular CLI can be installed using Node package manager using the command shown below

```
1. D:\> npm install -g @angular/cli
```

Test successful installation of Angular CLI using the following command

Note: Sometimes additional dependencies might throw an error during CLI installation but still check whether CLI is installed or not using the following command. If the version gets displayed, you can ignore the errors.

```
1. D:\> ng v
```

```
D:\>ng v

Angular CLI
Angular CLI: 13.1.2
Node: 16.13.0
Package Manager: npm 8.1.0
OS: win32 x64

Angular:
...

Package                                  Version
-----
@angular-devkit/architect                0.1301.2 (cli-only)
@angular-devkit/core                     13.1.2 (cli-only)
@angular-devkit/schematics               13.1.2 (cli-only)
@schematics/angular                     13.1.2 (cli-only)
```

Angular CLI is a command-line interface tool to build Angular applications. It makes application development faster and easier to maintain.

Using CLI, you can create projects, add files to them, and perform development tasks such as testing, bundling, and deployment of applications.

Command	Purpose
npm install -g @angular/cli	Installs Angular CLI globally
ng new <project name>	Creates a new Angular application
ng serve --open	Builds and runs the application on lite-server and launches a browser
ng generate <name>	Creates class, component, directive, interface, module, pipe, and service
ng build	Builds the application
ng update @angular/cli @angular/core	Updates Angular to a newer version

1.b Course Name: Angular JS

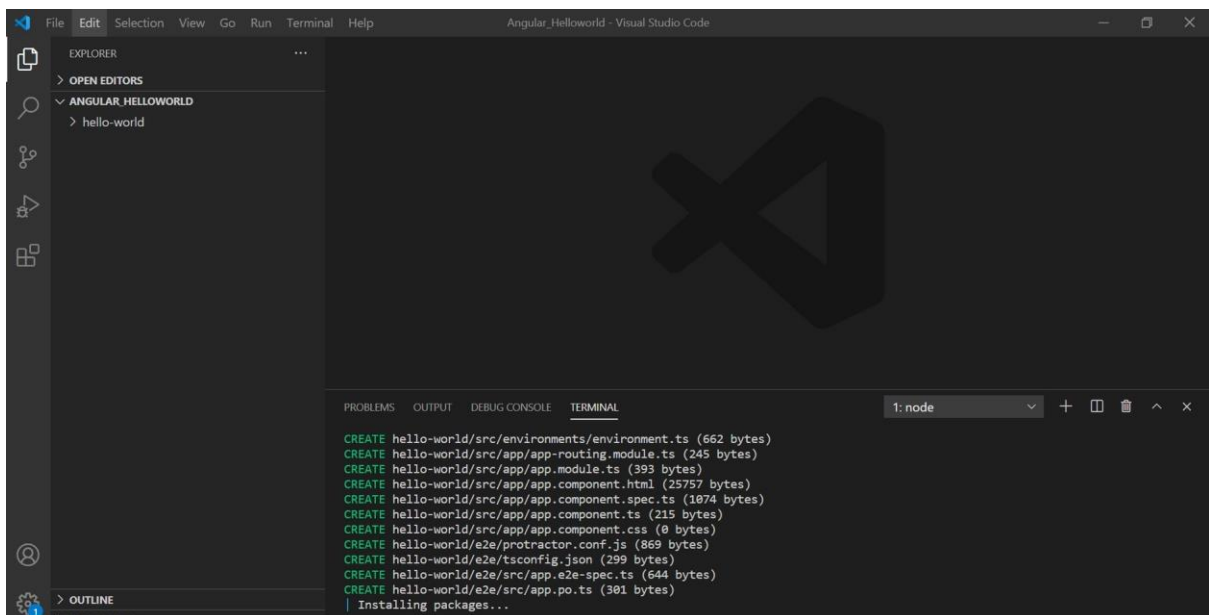
Module Name: Components and Modules

Create a new component called hello and render Hello Angular on the page Creating the Angular HelloWorld Application

Step 1

Create a folder for your application in the desired location on your system and open it on VSCode. Open a new terminal and type in the following command to create your app folder.

ng create hello-world



When the command is run, Angular creates a skeleton application under the folder. It also includes a bunch of files and other important necessities for the application.

Step 2

To run the application, change the directory to the folder created, and use the ng command.

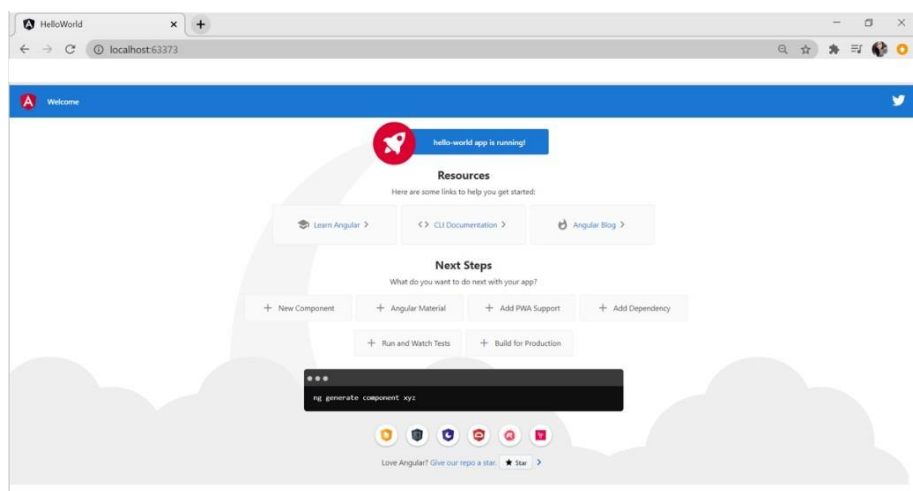
```
cd hello-world
```

```
ng serve
```

Once run, open your browser and navigate to localhost:4200. If another application is running on that address, you can simply run the command.

```
ng serve--port
```

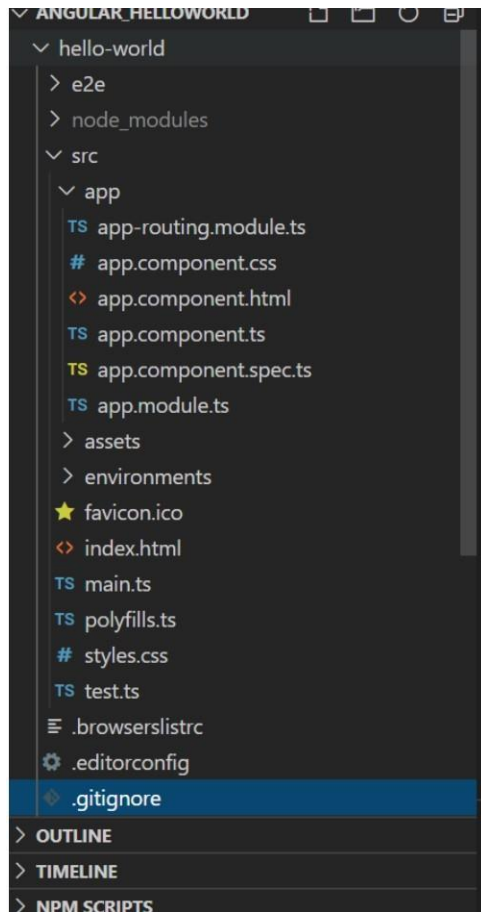
It will generate a port for you to navigate to. Typically, the browser looks something like this



You can leave the ng serve command running in the terminal, and continue making changes. If you have the application opened in your browser, it will automatically refresh each time you save your changes. This makes the development quick and iterative.

Basics of an Angular App

At its core, any Angular application is still a Single-Page Application (SPA), and thus its loading is triggered by a main request to the server. When we open any URL in our browser, the very first request is made to our server. This initial request is satisfied by an HTML page, which then loads the necessary JavaScript files to load both Angular as well as our application code and templates.



Root HTML - index.html

```
<!doctype html>
```

```
<html lang="en">
```

```
<head>
```

```
<meta charset="utf-8">
```

```
<title>HelloWorld</title>
```

```
<base href="/">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

```
<link rel="icon" type="image/x-icon" href="favicon.ico">
```

```
</head>
```

```
<body>
```

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

```
</body>
```

```
</html>
```

The root component looks very pristine and neat, with barely any references or dependencies. The only main thing in this file is the `<app-root>` element. This is the marker for loading the application. All the application code, styles, and inline templates are dynamically injected into the `index.html` file at run time by the `ng serve` command.

2.a Course Name: Angular JS

Module Name: Structural Directives - ngIf

Create a login form with username and password fields. If the user enters the correct credentials, it should render a "Welcome <<username>>" message otherwise it should render "Invalid Login!!! Please try again..." message

Simple Login Example in AngularJS

We have the main base page i.e. **index.html** where all other views are loaded. The following code explains the index page. Base script files that are required to run an application, are loaded when the application is bootstrapped.

index.html

```
1  | <!doctype
   | html>
2  | <html ng-
   | app="myApp" >
3  | <head>
4  |   <meta charset="utf
   | -8">
5  |   <title>AngularJS
   | Plunker</title>
6  |   <script>document.write('<base href="'
   | + document.location + "' />');</script>
7  |   <link href="//netdna.bootstrapcdn.com/twitter-
   | bootstrap/2.3.1/css/bootstrap-
   | combined.min.css" rel="stylesheet">
8  |   <link rel="stylesheet" href="style.css">
9  |   <script src="http://code.angularjs.org/1.2.13/angular.js"></script>
10 |   <script src="//cdnjs.cloudflare.com/ajax/libs/angular-
   | ui-router/0.2.8/angular-ui-router.min.js"></script>
11 |
12 |   <script src="app.js"></script>
13 |   <script src="loginController.js"></script>
14 |   <script src="homeController.js"></script>
15 |   <script src="loginService.js"></script>
```

```

16 | </head>
17 | <body>
18 |   <div class="container" width="100px">
19 |     <h2>AngularJS
      Simple Login
      Example</h2>
20 |
21 |     <div ui-
      view></div>
22 |   </div>
23 | </body>
24 | </html>

```

Now, we need a javascript to hold all the configuration details required for an application to run. In this file, we configure the routes for an application which is done using the UI Router.

app.js.

```

1 | (function()
  | {
2 |   var app =
      angular.module('myApp',
      ['ui.router']);
3 |
4 |   app.run(function($rootScope,
      $location, $state,
      LoginService) {
5 |     console.clear();
6 |     console.log('running');
7 |     if(!
      LoginService.isAuthenticated()) {
8 |       $state.transitionTo('login');
9 |     }
10 |   });
11 |
12 |   app.config(['$stateProvider', '$urlRouterProvider',

```



```

13 | function($stateProvider,
    | $urlRouterProvider) {
14 |     $stateProvider
15 |     .state('login',
    |     {
16 |         url : '/login',
17 |         templateUrl : 'login.html',
18 |         controller : 'LoginController'
19 |     })
20 |     .state('home',
    |     {
21 |         url : '/home',
22 |         templateUrl : 'home.html',
23 |         controller : 'HomeController'
24 |     });
25 |
26 |     $urlRouterProvider.otherwise('/login');
27 | });
28 |
29 | })();

```

Next, we will have the views to be loaded on the index page. So firstly it's the login page. In the login page, we have two simple fields to be entered. They are UserName and Password. Also, we have a login button.

login.html

```

1 | <div class="col-
    | md-12">
2 |     <p><strong>Login
    | Page</strong></p>
3 |
4 |     <form ng-submit="formSubmit()"
    | class="form">
5 |         <div class="col-
    | md-4">

```

```

6      <div class="form-
      | group">
7      |      <input type="text" class="form-control" ng-
      |      model="username"placeholder="username" required=""/>
8      |      </div>
9      |
10     |      <div class="form-
      | group">
11     |      <input type="password" class="form-control" ng-
      |      model="password"placeholder="password" required=""/
      |      >
12     |      </div>
13     |
14     |      <div class="form-
      | group">
15     |      <button type="submit" class="btn
      | btn-success">Login</button>
16     |      <span class="text-danger" style="colo
      | r:red">{{ error }}</span>
17     |      </div>
18     |
19     |      </div>
20     |      </form>
21     | </div>

```

The login page has its own controller to hold the business logic. This is a simple javascript file created specifically for **login.html** where all its functions and logic are included.

loginController.js

```

1      | var app =
      | angular.module('myApp');
2      | app.controller('LoginController', function($scope,
      | $rootScope, $stateParams, $state, LoginService) {
3      |     $rootScope.title
      |     = "AngularJS
      |     Login Sample";
4      |

```

```

5      $scope.formSubmit
      = function() {

6          if(LoginService.login($scope.username,
            $scope.password)) {

7              $rootScope.userName
              = $scope.username;

8              $scope.error
              = "";

9              $scope.username
              = "";

10             $scope.password
              = "";

11             $state.transitionTo('home');

12         } else {

13             $scope.error
              = "Incorrect
              username/password !";

14         }

15     };

16 });

```

Next, we will implement the home page (**home.html**) which will be loaded once the user is successfully logged in. Here, we have simple statements and the logout button to sign out from an application.

home.html

```

1      <div class="col-md-12" st
      yle="width:500px;">

2          <div align="right"><a ui-
            sref="login">Logout</a></div>

3          <h4>Welcome
            {{user}}! </h4>

4          <p><strong>This
            is Home
            Page</strong></p>

5      </div>

```

As we had a separate controller for the login page, similarly we will have a separate controller for the home page. You can see the code for the controller page below.

homeController.js

```

1 | var app =
  | angular.module('myApp');
2 |
3 | app.controller('HomeController',
4 |   function($scope,
  |     $rootScope,
  |     $stateParams,
  |     $state,
  |     LoginService) {
5 |     $scope.user =
  |     $rootScope.userName;
6 |   });

```

Since we are showing the username of who is logging in to the application, we are storing it in the scope object and binding it to the variable.

As we have said above, we authenticate the user after the user enters credentials and clicks on the login button. This is done in the **loginService.js** file.

You can see the implementation details for **loginService.js** as shown below. We have preassumed the user credentials to be admin and password. We need to make use of the database server to authenticate the user with his/her registered credentials.

loginService.js

```

1 | var app =
  | angular.module('myApp');
2 |
3 | app.factory('LoginService',
  | function() {
4 |   var
  |   admin
  |   =
  |   'admin';
5 |   var pass
  |   =
  |   'password';
6 |   var
  |   isAuthenticated
  |   = false;
7 |
8 |   return
  |   {

```

```

9      login :
      function(username,
        password) {

10          isAuthenticated
            = username ===
              admin &&
                password === pass;

11          return
            isAuthenticated;

12      },

13      isAuthenticated :
      function() {

14          return
            isAuthenticated;

15      }

16  };

17  });

```

Now you might be thinking why we have done this logic in the service file, we could have done it in the loginController page itself. So to answer this thought, Yes, you are perfectly right. We could have done it in the loginController itself. But, one important thing is, as you know services are created to separate the logic from the controller so that it can be reused. In the same way, we have created a separate service to implement the authentication logic so that it can be reused when required.

The outputs of the login application are shown below.

Below the login form is displayed initially when the application is loaded.



This is how the login form looks like

If the user does not enter the fields and tries to click on the login button, the page notifies him about the required fields. It gives a message as shown in the below image.

Field validation

If the user enters the wrong credentials, the error message is displayed in red color as shown below.

Invalid credentials message

If the user enters the correct credentials, the user is authenticated. Once the authentication is successful, the user will be redirected to the home page as shown below.

If the authentication fails, he will remain on the login page and an error message is displayed on the page.

The home page has a Logout link to log out from an application. The home page looks as shown below.

User's home page

2.b Course Name: Angular JS

Module Name: ngFor

Create a courses array and rendering it in the template using ngFor directive in a list format.

NgFor is a structural directive, meaning that it changes the structure of the DOM.

It's point is to repeat a given HTML template once for each value in an array, each time passing it the array value as context for string interpolation or binding.

Sometimes we also want to get the *index* of the item in the array we are iterating over.

We can do this by adding another variable to our `ngFor` expression and making it equal to `index`, like so:

HTML

Copy (1)

```
<li *ngFor="let person of people; let i = index"> (1)
  {{ i + 1 }} - {{ person.name }} (2)
</li>
</ul>
```

Here is the code to create a `courses` array and rendering it in the template using `ngFor`:

```
// app.component.ts

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

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  courses = [
    {
      id: 1,
      name: 'Angular',
      description: 'A web framework for building single-page applications'
    },
    {
      id: 2,
      name: 'React',
      description: 'A JavaScript library for building user interfaces'
    },
    {
      id: 3,
      name: 'Vue',
      description: 'A JavaScript framework for building user interfaces'
    }
  ];
}

// app.component.html

<h1>Courses</h1>
<ul *ngFor="let course of courses">
  <li>{{ course.name }}</li>
  <li>{{ course.description }}</li>
</ul>
```

Use code with caution.

[Learn more](#)

This code will create an array of `courses` with three objects. Each object will have an `id`, `name`, and `description`. The template will use the `ngFor` directive to iterate over the `courses` array and render each course in a list.

3.a Course Name: Angular JS

Module Name: Attribute Directives - ngStyle

Apply multiple CSS properties to a paragraph in a component using ngStyle.

The `NgStyle` directive lets you set a given DOM elements style properties.

One way to set styles is by using the `NgStyle` directive and assigning it an *object literal*, like so:

HTML

```
<div [ngStyle]="{'background-color':'green'}"></div>
```

This sets the background color of the `div` to green.

`ngStyle` becomes much more useful when the value is *dynamic*. The *values* in the object literal that we assign to `ngStyle` can be JavaScript expressions which are evaluated and the result of that expression is used as the value of the CSS property, like this:

HTML

```
<div [ngStyle]="{'background-color':person.country === 'UK' ? 'green' : 'red' }"></div>
```

The above code uses the ternary operator to set the background color to green if the persons country is the UK else red.

But the expression doesn't have to be *inline*, we can call a function on the component instead.

To demonstrate this let's flesh out a full example. Similar to the ones we've created before let's loop through an array of people and print out their names in different colors depending on the country they are from.

TypeScript

```
@Component({
  selector: 'ngstyle-example',
  template: `<h4>NgStyle</h4>
<ul *ngFor="let person of people">
  <li [ngStyle]="{'color':getColor(person.country)}"> {{ person.name }} ({{ person.country }}) (1)
</li>
</ul>
`
})
class NgStyleExampleComponent {

  getColor(country) { (2)
    switch (country) {
      case 'UK':
        return 'green';
      case 'USA':
        return 'blue';
      case 'HK':
        return 'red';
    }
  }

  people: any[] = [
    {
      "name": "Douglas Pace",
      "country": 'UK'
    },
    {
      "name": "Mcleod Mueller",
      "country": 'USA'
    },
    {
      "name": "Day Meyers",
      "country": 'HK'
    },
    {
      "name": "Aguirre Ellis",
```

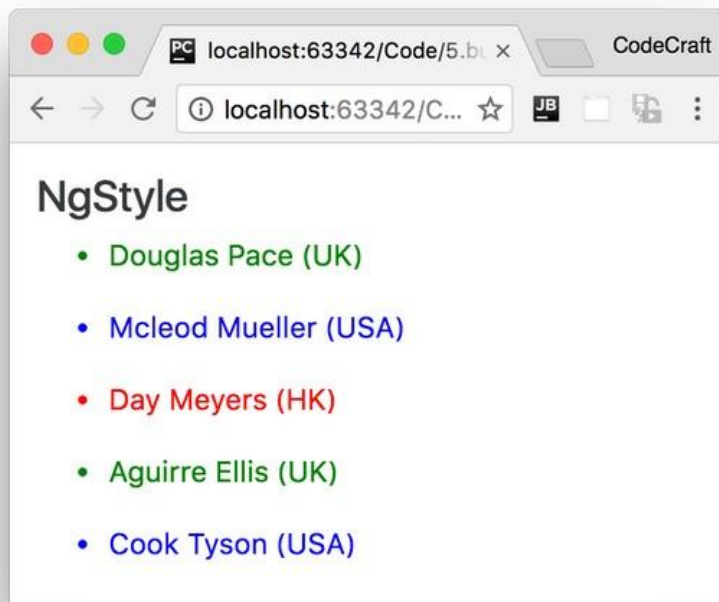
```

    "country": 'UK'
  },
  {
    "name": "Cook Tyson",
    "country": 'USA'
  }
];
}

```

We set the color of the text according to the value that's returned from the `getColor` function.

Our `getColor` function returns different colors depending on the country passed in. If we ran the above code we would see:



3.b Course Name: Angular JS

Module Name: ngClass

Apply multiple CSS classes to the text using ngClass directive.

To add multiple CSS classes in Angular you can use the

ngClass

directive. The

ngClass

directive is used to add a single or multiple class to an element dynamically(condition based).

In order to add multiple classes using the

ngClass

directive, you have to separate each pair of the class and condition with a comma and you can add as many classes as you want.
This is how you can do it:

```
<div [ngClass]="{'class1': condition1, 'class2': condition2, ...}"></div>
```

The *class1* will be added to the div element if *condition1* evaluates to true, *class2* will be added if *condition2* evaluates to true and so on.

Example:

Let's say we want to add two classes

highlight

and

bold

to a div element based on the marks of the student.

If the student's marks are greater than 80, we want to highlight the text and if the marks are greater than 90, we also want to make the text bold.

This is what you need to add to your template file:

```
<div [ngClass]="{'highlight': marks>80, 'bold': marks>90}">John Doe</div>
```

Secondly, you have to declare the variable

marks

inside your ts file. In real scenarios, the variable

marks

will hold a dynamic value.

```
// Declare variable marks
```

```
marks: number = 95;
```

Next, you need to define both classes inside your component's CSS file:

```
.highlight{  
  background: yellow;  
}  
.bold{  
  font-weight: bold;  
}
```

The `NgClass` directive allows you to set the CSS class dynamically for a DOM element.

Tip

The `NgClass` directive will feel very similar to what `ngClass` used to do in Angular 1.

There are two ways to use this directive, the first is by passing an object literal to the directive, like so:

HTML

```
[ngClass]="{'text-success':true}"
```

When using an object literal, the keys are the classes which are added to the element if the value of the key evaluates to true.

So in the above example, since the value is `true` this will set the class `text-success` onto the element the directive is attached to.

The value can also be an *expression*, so we can re-write the above to be.

HTML

```
[ngClass]='{'text-success':person.country === 'UK'}'
```

Let's implement the colored names demo app using `ngClass` instead of `ngStyle`.

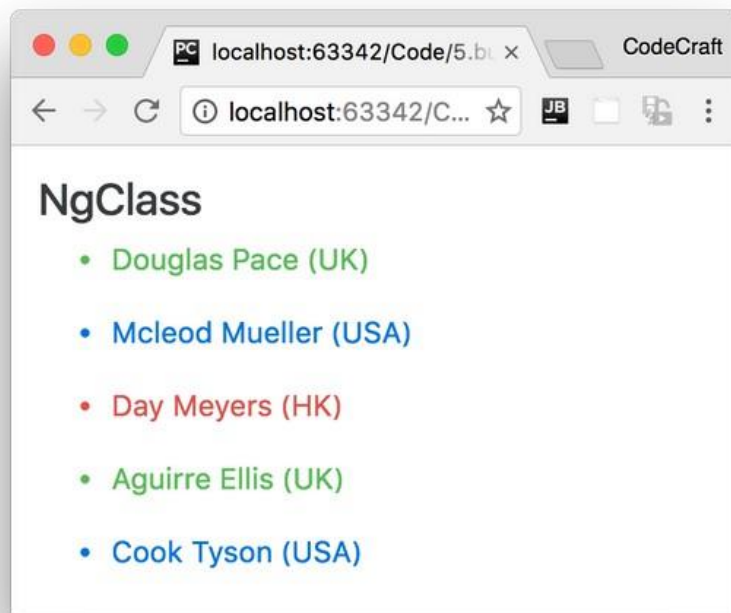
HTML

```
<h4>NgClass</h4>
<ul *ngFor="let person of people">
  <li [ngClass]='{'
    'text-success':person.country === 'UK',
    'text-primary':person.country === 'USA',
    'text-danger':person.country === 'HK'
  }">{{ person.name }} ({{ person.country }})
</li>
</ul>
```

Since the object literal can contain many keys we can also set many class names.

We can now color our text with different colors for each country with one statement.

If we ran the above code we would see:



4.a Course Name: Angular JS

Module Name: Property Binding

Binding image with class property using property binding.

Binding to a property

To bind to an element's property, enclose it in square brackets, [], which identifies the property as a target property.

A target property is the DOM property to which you want to assign a value.

To assign a value to a target property for the image element's src property, type the following code:

```
src/app/app.component.html
```

```
content_copy<img alt="item" [src]="itemImageUrl">
```

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute.

In this example, src is the name of the `` element property.

The brackets, [], cause Angular to evaluate the right-hand side of the assignment as a dynamic expression.

Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

To assign a string to a property, type the following code:

```
src/app.component.html
```

```
content_copy<app-item-detail childItem="parentItem"></app-item-detail>
```

Omitting the brackets renders the string parentItem, not the value of parentItem.

Setting an element property to a component property value

To bind the src property of an `` element to a component's property, place src in square brackets followed by an equal sign and then the property.

Using the property imageUrl, type the following code:

```
src/app/app.component.html
```

```
content_copy<img alt="item" [src]="itemImageUrl">
```

Declare the itemImageUrl property in the class, in this case AppComponent.

```
src/app/app.component.ts
```

```
content_copyitemImageUrl = '../assets/phone.svg';
```

colspan and colSpan

A common point of confusion is between the attribute, colspan, and the property, colSpan. Notice that these two names differ by only a single letter.

To use property binding using colSpan, type the following:

```
src/app/app.component.html
```

```
content_copy<!-- Notice the colSpan property is camel case -->
```

```
<tr><td [colSpan]="1 + 1">Three-Four</td></tr>
```

To disable a button while the component's isUnchanged property is true, type the following:

```
src/app/app.component.html
```

```
content_copy<!-- Bind button disabled state to `isUnchanged` property -->
```

```
<button type="button" [disabled]="isUnchanged">Disabled Button</button>
```

To set a property of a directive, type the following:

```
src/app/app.component.html
```

```
content_copy<p [ngClass]="classes">[ngClass] binding to the classes property making this blue</p>
```

To set the model property of a custom component for parent and child components to communicate with each other, type the following:

```
src/app/app.component.html
```

```
content_copy<app-item-detail [childItem]="parentItem"></app-item-detail>
```

Toggle button features

To use a Boolean value to disable a button's features, bind the disabled DOM attribute to a Boolean property in the class.

```
src/app/app.component.html
```

```
content_copy<!-- Bind button disabled state to `isUnchanged` property -->
```

```
<button type="button" [disabled]="isUnchanged">Disabled Button</button>
```

Because the value of the property isUnchanged is true in the AppComponent, Angular disables the button.

```
src/app/app.component.ts
```

```
content_copyisUnchanged = true;
```

4.b Course Name: Angular JS

Module Name: Attribute Binding

Binding colspan attribute of a table element to the class property.

Binding to colspan

Another common use case for attribute binding is with the colspan attribute in tables. Binding to the colspan attribute helps you to keep your tables programmatically dynamic. Depending on the amount of data that your application populates a table with, the number of columns that a row spans could change.

To use attribute binding with the <td> attribute colspan

1. Specify the colspan attribute by using the following syntax: [attr.colspan].
2. Set [attr.colspan] equal to an expression.

In the following example, you bind the colspan attribute to the expression 1 + 1.

src/app/app.component.html

```
content_copy<!-- expression calculates colspan=2 -->

<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```

This binding causes the <tr> to span two columns.

Sometimes there are differences between the name of property and an attribute.

colspan is an attribute of <td>, while colSpan with a capital "S" is a property. When using attribute binding, use colspan with a lowercase "s".

For more information on how to bind to the colSpan property, see the colspan and colSpan section of [Property Binding](#).

In Angular, you can bind to the colspan attribute of a table element using the following syntax:

1. Specify the colspan attribute using the syntax [attr.colspan]
2. If you want to bind to an attribute, use the syntax [attr.colspan]="count"

The colspan attribute helps you keep your tables programmatically dynamic. The number of columns that a row spans may vary depending on the amount of data that your application populates a table with.

In computing, an attribute is a specification that defines a property of an object, element, or file. An attribute of an object usually consists of a name and a value. For an element, these can be a type and class name.

In Angular, a binding creates a live connection between a part of the UI created from a template and the model.

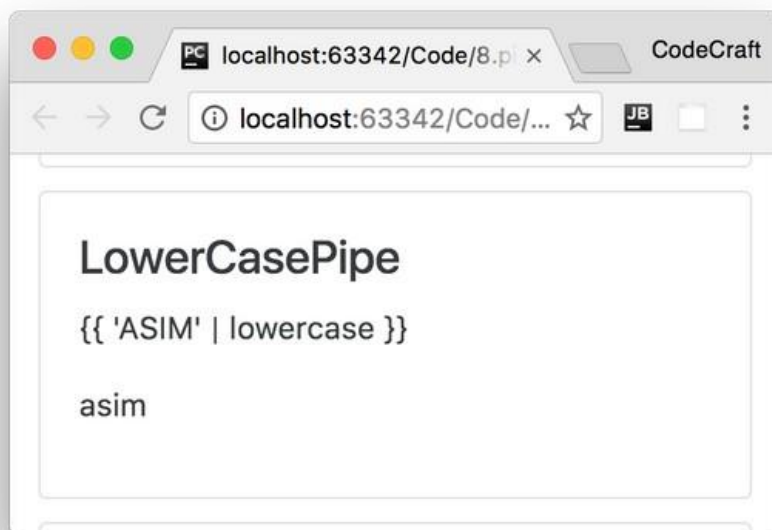
5.a Course Name: Angular JS

Module Name: Built in Pipes

Display the product code in lowercase and product name in uppercase using built-in pipes.

LowerCasePipe

This transforms a string to lowercase, like so:



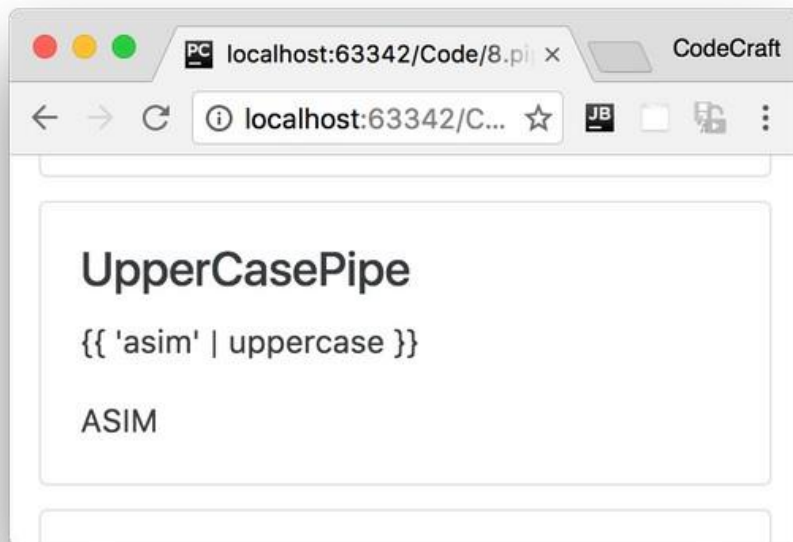
HTML

```
Copy<div class="card card-block">
  <h4 class="card-title">LowerCasePipe</h4>
  <div class="card-text">
    <p ngNonBindable>{{ 'ASIM' | lowercase }}</p>
    <p>{{ 'ASIM' | lowercase }}</p>
  </div>
```


</div>

UpperCasePipe

This transforms a string to uppercase, like so:



HTML

Copy<div class="card card-block">

```
<h4 class="card-title">UpperCasePipe</h4>
<div class="card-text">
  <p ngNonBindable>{{ 'asim' | uppercase }}</p>
  <p>{{ 'asim' | uppercase }}</p>
</div>
</div>
```

5.b Course Name: Angular JS

Module Name: Passing Parameters to Pipes

Apply built-in pipes with parameters to display product details.

To apply built-in pipes with parameters to display product details in Angular, you can use the following steps:

1. Register the pipe in the module
2. Inject the pipe in the component
3. Call the pipes' transform method
4. Use the pipe operator (|) within a template expression
5. Use the name of the pipe
6. Apply a minimum number of 3 to the value 1

Angular comes with a set of pre-built pipes to handle most of the common transformations. For example, you can use pipes to:

- Transform strings, currency amounts, dates, and other data for display
- Render numbers and string values in a locale-specific format
- Get a slice of arrays or strings
- Get values returned from promises and observables

You can also **create custom pipes to encapsulate transformations that are not provided with the built-in pipes.**

Example: Transforming a value exponentially

In a game, you might want to implement a transformation that raises a value exponentially to increase a hero's power. For example, if the hero's score is 2, boosting the hero's power exponentially by 10 produces a score of 1024. Use a custom pipe for this transformation.

The following code example shows two component definitions:

- The `exponential-strength.pipe.ts` component defines a custom pipe named `exponentialStrength` with the `transform` method that performs the transformation. It defines an argument to the `transform` method (`exponent`) for a parameter passed to the pipe.
- The `power-booster.component.ts` component demonstrates how to use the pipe, specifying a value (2) and the `exponent` parameter (10).

`src/app/exponential-strength.pipe.ts`
`src/app/power-booster.component.ts`

```
content_copyimport { Pipe, PipeTransform } from '@angular/core';

/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
 */

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {

    return Math.pow(value, exponent);

  }

}
```

The browser displays the following:

```
content_copyPower Booster
```

```
Superpower boost: 1024
```

To examine the behavior of the exponentialStrength pipe in the [live example](#) / [download example](#), change the value and optional exponent in the template.

6.a Course Name: Angular JS

Module Name: Passing data from Container Component to Child Component

Create an AppComponent that displays a dropdown with a list of courses as values in it. Create another component called the CoursesList component and load it in AppComponent which should display the course details.

Here is the code for the CoursesList component:

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

@Component({
  selector: 'app-courses-list',
  templateUrl: './courses-list.component.html',
  styleUrls: ['./courses-list.component.css']
})
export class CoursesList implements OnInit {
  courses = [
    {
      name: 'Angular',
      description: 'A web framework for building single-page applications.'
    },
    {
      name: 'React',
      description: 'A JavaScript library for building user interfaces.'
    },
    {
      name: 'Vue',
      description: 'A JavaScript framework for building user interfaces.'
    }
  ];
  constructor() {}
}
```

```

ngOnInit() {
}
}

```

To use these components, you would first need to import them into your Angular application. Then, you could add the AppComponent to your HTML code as follows:

```
<my-app></my-app>
```

This would create an instance of the AppComponent, which would in turn display the list of courses. You could then use the CoursesList component to display the details of a particular course. To do this, you would simply need to add the CoursesList component to your HTML code as follows:

```
<app-courses-list></app-courses-list>
```

This would create an instance of the CoursesList component, which would in turn display the details of the currently selected course.

6.b Course Name: Angular JS

Module Name: Passing data from Child Component to ContainerComponent

Create an AppComponent that loads another component called the CoursesList component. Create another component called CoursesListComponent which should display the courses list in a table along with a register .button in each row.

Creating a component

The best way to create a component is with the Angular CLI. You can also create a component manually.

Creating a component using the Angular CLI

To create a component using the Angular CLI:

1. From a terminal window, navigate to the directory containing your application.
2. Run the ng generate component <component-name> command, where <component-name> is the name of your new component.

By default, this command creates the following:

- A directory named after the component
- A component file, <component-name>.component.ts
- A template file, <component-name>.component.html
- A CSS file, <component-name>.component.css
- A testing specification file, <component-name>.component.spec.ts

Where <component-name> is the name of your component.

You can change how ng generate component creates new components. For more information, see ng generate component in the Angular CLI documentation.

Creating a component manually

Although the Angular CLI is the best way to create an Angular component, you can also create a component manually. This section describes how to create the core component file within an existing Angular project.

To create a new component manually:

1. Navigate to your Angular project directory.
2. Create a new file, <component-name>.component.ts.
3. At the top of the file, add the following import statement.

```
content_copyimport { Component } from '@angular/core';
```

4. After the import statement, add a @Component decorator.

```
5. content_copy@Component({  
  
  })
```

6. Choose a CSS selector for the component.

```
7. content_copy@Component({  
  
8.   selector: 'app-component-overview',  
  
  })
```

For more information on choosing a selector, see [Specifying a component's selector](#).

9. Define the HTML template that the component uses to display information. In most cases, this template is a separate HTML file.

```
10. content_copy@Component({  
  
11.   selector: 'app-component-overview',  
  
12.   templateUrl: './component-overview.component.html',  
  
  })
```

For more information on defining a component's template, see [Defining a component's template](#).

13. Select the styles for the component's template. In most cases, you define the styles for your component's template in a separate file.

```
14. content_copy@Component({  
  
15.   selector: 'app-component-overview',  
  
16.   templateUrl: './component-overview.component.html',  
  
17.   styleUrls: ['./component-overview.component.css']  
  
  })
```

18. Add a class statement that includes the code for the component.

```
19. content_copyexport class ComponentOverviewComponent {  
  
20.  
  
  }
```

Specifying a component's CSS selector

Every component requires a CSS *selector*. A selector instructs Angular to instantiate this component wherever it finds the corresponding tag in template HTML. For example, consider a component `hello-world.component.ts` that defines its selector as `app-hello-world`. This selector instructs Angular to instantiate this component any time the tag `<app-hello-world>` appears in a template.

Specify a component's selector by adding a `selector` property to the `@Component` decorator.

```
content_copy@Component({  
  selector: 'app-component-overview',  
})
```

Defining a component's template

A template is a block of HTML that tells Angular how to render the component in your application. Define a template for your component in one of two ways: by referencing an external file, or directly within the component.

To define a template as an external file, add a `templateUrl` property to the `@Component` decorator.

```
content_copy@Component({  
  selector: 'app-component-overview',  
  templateUrl: './component-overview.component.html',  
})
```

To define a template within the component, add a `template` property to the `@Component` decorator that contains the HTML you want to use.

```
content_copy@Component({  
  selector: 'app-component-overview',  
  template: '<h1>Hello World!</h1>',  
})
```

If you want your template to span multiple lines, use backticks (```). For example:

```
content_copy@Component({  
  selector: 'app-component-overview',  
  template: `  
    <h1>Hello World!</h1>  
    <p>This template definition spans multiple lines.</p>`  
})
```

An Angular component requires a template defined using `template` or `templateUrl`. You cannot have both properties in a component.

Declaring a component's styles

Declare component styles used for its template in one of two ways: By referencing an external file, or directly within the component.

To declare the styles for a component in a separate file, add a `styleUrls` property to the `@Component` decorator.

```
content_copy@Component({  
  selector: 'app-component-overview',  
  templateUrl: './component-overview.component.html',  
  styleUrls: ['./component-overview.component.css']  
})
```

To declare the styles within the component, add a `styles` property to the `@Component` decorator that contains the styles you want to use.

```
content_copy@Component({  
  selector: 'app-component-overview',  
  template: '<h1>Hello World!</h1>',  
  styles: ['h1 { font-weight: normal; }']  
})
```

The `styles` property takes an array of strings that contain the CSS rule declarations.

7.a Course Name: Angular JS

Module Name: Template Driven Forms

Create a course registration form as a template-driven form.

Here is the code for a course registration form as a template-driven form:

```
<form (ngSubmit)="onSubmit()">  
  <h1>Course Registration</h1>  
  
  <div class="form-group">  
    <label for="name">Name</label>  
    <input type="text" class="form-control" id="name" [(ngModel)]="name" required>  
  </div>  
  
  <div class="form-group">  
    <label for="email">Email</label>  
    <input type="email" class="form-control" id="email" [(ngModel)]="email" required>  
  </div>  
  
  <div class="form-group">  
    <label for="course">Course</label>  
    <select class="form-control" id="course" [(ngModel)]="course" required>  
      <option value="">Select a course</option>  
      <option value="angular">Angular</option>  
      <option value="react">React</option>  
      <option value="vue">Vue</option>  
    </select>  
  </div>  
</form>
```

```
<button type="submit" class="btn btn-primary">Register</button>
</form>
```

The code above creates a form with three fields: name, email, and course. The name and email fields are required, and the course field must be selected. The form submits the data to the component class when the user clicks the Register button.

Here is the code for the component class that handles the form submission:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-course-registration',
  templateUrl: './course-registration.component.html',
  styleUrls: ['./course-registration.component.css']
})
export class CourseRegistrationComponent implements OnInit {

  form: FormGroup;

  constructor(private fb: FormBuilder) {
    this.form = this.fb.group({
      name: ['', Validators.required],
      email: ['', Validators.required],
      course: ['', Validators.required]
    });
  }

  ngOnInit() {
  }

  onSubmit() {
    // TODO: Submit the form data to the backend.
  }
}
```

The code above creates a FormGroup object that represents the form. The FormGroup object is bound to the form in the template using the [(ngModel)] directive. The form data is submitted to the component class when the user clicks the Register button.

7.b Course Name: Angular JS

Module Name: Model Driven Forms or Reactive Forms

Create an employee registration form as a reactive form.

Creating a User Registration UI Using Reactive Forms

Let's start by creating an Angular app from scratch and see how to use Reactive Forms in Angular 7.

Assuming you have installed the Angular CLI on your system, let's create an Angular app using the Angular CLI command.

```
ng new form-app
```

The above command creates an Angular project with some boilerplate code. Let's take a look at the boilerplate code.

Inside your form-app/src/app folder, you'll find the default AppComponent. Let's remove the default component from the project. Remove all files from the app folder except the app.module.ts file.

Once you have removed the default component, let's create a root component for the Angular app.

ng generate component root

Update the app.module.ts file to remove the AppComponent and update the bootstrap property in NgModule to RootComponent.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
```

```
import { RootComponent } from './root/root.component';
```

```
@NgModule({
  declarations: [
    RootComponent
  ],
  imports: [
    BrowserModule
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [RootComponent]
})
export class AppModule { }
```

Save the above changes and try running the Angular app.

npm start

You'll be able to see the default app running.

Let's start creating the form UI. You'll be making use of the FormBuilder to create a Reactive form. Inside the RootComponent class, import the FormBuilder module.

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

Instantiate the FormBuilder module inside the RootComponent constructor.

```
constructor(private formBuilder: FormBuilder) { }
```

Using the FormBuilder module, create a form group that will define the fields in the form.

```
ngOnInit() {
  this.userForm = this.formBuilder.group({
    firstName: [''],
    lastName: [''],
    email: [''],
    password: ['']
  });
}
```

Once you have defined the form builder group, set up the RootComponent form with the form group userForm. Add the formGroup directive to the form inside the root.component.html file.

```
<form [formGroup]="userForm" class="text-center border border-light p-5">
```

```
<p class="h4 mb-4">Sign up</p>
```

```
<div class="form-row mb-4">
  <div class="col">
    <input type="text" id="defaultRegisterFormFirstName" class="form-control" placeholder="First
name">
  </div>
```

```

    <div class="col">
      <input type="text" id="defaultRegisterFormLastName" class="form-control" placeholder="Last name">
    </div>
  </div>

```

```

    <input type="email" id="defaultRegisterFormEmail" class="form-control mb-4" placeholder="E-mail">

```

```

    <button class="btn btn-info my-4 btn-block" type="submit">Sign in</button>

```

```

  <hr>

```

```

</form>

```

Add the formControlName to each element in the form, as in the userGroup.

```

<form [formGroup]="userForm" (ngSubmit)="onSubmit()" class="text-center border border-light p-5">

```

```

  <p class="h4 mb-4">Sign up</p>

```

```

  <div class="form-row mb-4">
    <div class="col">
      <input type="text" formControlName="firstName" id="defaultRegisterFormFirstName" class="form-control" placeholder="First name">
    </div>
    <div class="col">
      <input type="text" formControlName="lastName" id="defaultRegisterFormLastName" class="form-control" placeholder="Last name">
    </div>
  </div>

```

```

  <input type="email" formControlName="email" id="defaultRegisterFormEmail" class="form-control mb-4" placeholder="E-mail">

```

```

  <button class="btn btn-info my-4 btn-block" type="submit">Sign in</button>

```

```

  <hr>

```

```

</form>

```

Save the above changes and point your browser to <http://localhost:4200/>.

You will be able to see the user form.

8.a Course Name: Angular JS

Module Name: Custom Validators in Template Driven forms

Create a custom validator for the email field in the course registration form.

Email Validation in Angular

Angular provides several ways to validate emails, some of which are mentioned below:

- **Built-in validation:** Angular comes with some built-in email validators you can use to ensure the correctness and completeness of user-provided email addresses.
- **Pattern validation:** This allows you to specify a regular expression (regex) Angular email validation pattern that should match the user-provided value before validation can occur.
- **Custom validation:** You can also create your own custom email validators, especially if the built-in validators do not match your specific use case.

In this Angular email validation tutorial, we'll see how you can use these validation criteria to enhance the accuracy of user-provided email accounts.

How to Create Email Fields in Angular

Just like any other form element, Angular offers two different techniques for creating emails and processing the user input: reactive and template-driven.

Although the two approaches allow you to create intuitive email fields and embed them in your application, they differ in philosophies and programming styles. For the rest of this tutorial, we'll assume that you have some basic knowledge of how to use the two form modules.

Now, let's start getting our hands dirty...

Angular Email Validation Project Setup

Let's start by installing the Angular CLI, which we'll use to create a simple application for illustrating how to validate emails in Angular.

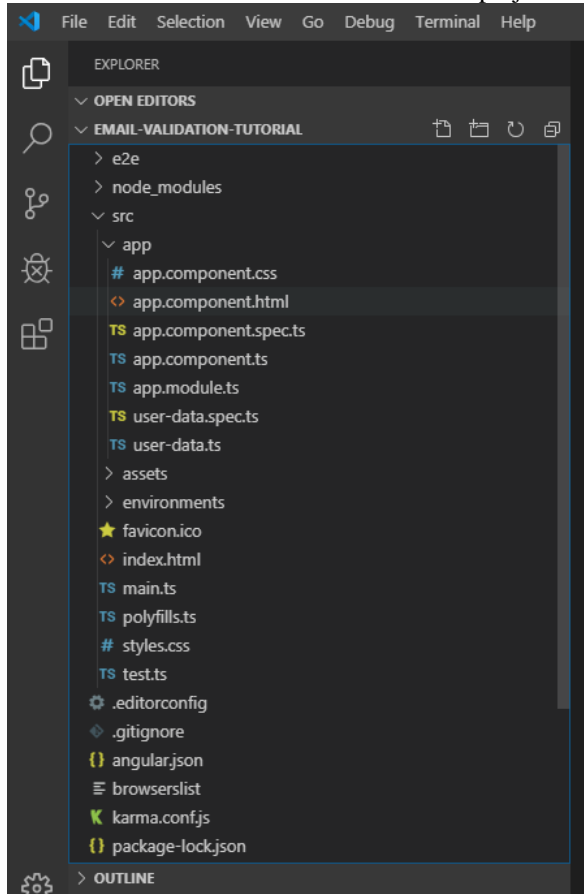
Here is the code to run on the command line interface:

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

Then, let's use the CLI to create a simple application where users can input their email addresses:

```
ng new email-validation-tutorial
```

Here is the structure of our email validation project on the Visual Studio Code editor:



To assist us in displaying useful validation and error messages to users, we'll include the Bootstrap CSS framework in our project. We'll add the Bootstrap CDN in the *index.html* file, inside the `<head>`.

Bootstrap comes with super-handful classes we can use to provide real-time feedback to users and assist them in providing accurate inputs. Remember that you can also create your own custom CSS classes to help in displaying error messages to users.

Next, in the *app.component.html* file, let's remove the default code and replace it with the following code for the email fields:

```
<!--app.component.html-->
<div class="container-fluid">
<form>
  <h2>Please provide your email address</h2>

  <div class="form-group">
    <label>Primary Email:</label>
    <input type="email" class="form-control" />
  </div>

  <div class="form-group">
    <label>Secondary Email:</label>
    <input type="email" class="form-control" />
  </div>
</form>
</div>
```

Here is how the email fields look on a browser:

Email Validation Tutorial

localhost:4200

Please provide your email addresses

Primary Email:

Secondary Email:

As you can see in the code above, we've just used HTML together with some Bootstrap classes to create the simple email form fields. We'll start adding some Angular code in the next steps.

Creating Reactive Forms in Angular

In Angular reactive forms, the component class is the main source of truth. Therefore, rather than implementing validation via attributes in the template—just like in template-driven situations—the validator functions are included in the form control model of the component class. This way, Angular can call these validators anytime there is a change in the value of the form control.

To add reactive capabilities to the email field, inside the *app.module.ts* file, let's import *ReactiveFormsModule* from the *@angular/forms* package and specify it in the imports array.

Here is the code:

```
//app.module.ts
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  imports: [
    //other imports here
    ReactiveFormsModule
  ],
  //more code here
})
export class AppModule { }
```

The *FormGroup* and the *FormControl* classes are some of the fundamental blocks for defining reactive forms in Angular applications. Therefore, we'll import them inside the *app.component.ts* file. A form group is essential for monitoring the form's validity status and changes in its child *FormControl* instances.

To create a new *FormGroup* instance, we'll define a constructor in the *AppComponent* class; it'll have an object of named keys that are mapped to their control.

The first argument passed in the *FormControl* constructor sets the initial value of the form element. In this case, we specified the initial values as empty strings to make the email fields appear empty on the UI.

So far, here is the code for the *app.component.ts* file:

```
//app.component.ts
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'email-validation-tutorial';
  userEmails = new FormGroup({
    primaryEmail: new FormControl(""),
    secondaryEmail: new FormControl("")
  });
}
```

```
}
```

Next, let's associate the FormGroup model with its view to create a communication layer between the two components.

Here is the updated code for the *app.component.html* file.

```
<!--app.component.html-->
<div class="container-fluid">
  <form [formGroup]="userEmails">

    <h3>Please provide your email addresses</h3>

    <div class="form-group">
      <label>Primary Email:</label>
      <input type="email" class="form-control" formControlName="primaryEmail" />
    </div>

    <div class="form-group">
      <label>Secondary Email:</label>
      <input type="email" class="form-control" formControlName="secondaryEmail" />
    </div>

  </form>
</div>
```

Consequently, the form control and the DOM can interact with one another: any changes in the model are reflected in the view, and vice versa.

8.b Course Name: Angular JS

Module Name: Services Basics

Create a Book Component which fetches book details like id, name and displays them on the page in a list format. Store the book details in an array and fetch the data using a custom service.

Certainly! Below is an example of a simple Book Component written in React that fetches book details using a custom service and displays them in a list format.

```
``jsx
import React, { useState, useEffect } from 'react';

// Custom service to fetch book details
const fetchBookDetails = async () => {
  try {
    // Assuming your API endpoint for book details is "/api/books"
    const response = await fetch('/api/books');
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching book details:', error);
    throw error;
  }
};
```

```

const BookComponent = () => {
  const [books, setBooks] = useState([]);

  useEffect(() => {
    const getBookDetails = async () => {
      try {
        const bookDetails = await fetchBookDetails();
        setBooks(bookDetails);
      } catch (error) {
        // Handle error, e.g., display an error message
      }
    };

    getBookDetails();
  }, []); // Empty dependency array ensures that useEffect runs only once, similar to componentDidMount

  return (
    <div>
      <h2>Book List</h2>
      <ul>
        {books.map((book) => (
          <li key={book.id}>
            <strong>ID:</strong> {book.id}, <strong>Name:</strong> {book.name}
          </li>
        ))}
      </ul>
    </div>
  );
};

export default BookComponent;

```

In this example:

- The `fetchBookDetails` function is a custom service that uses the Fetch API to make a request to the `/api/books` endpoint. You should replace this with your actual API endpoint.
- The `BookComponent` is a functional component that uses the `useState` and `useEffect` hooks from React.
- In the `useEffect` hook, it calls the `fetchBookDetails` function and updates the state with the fetched book details.
- The component then renders a list of books using the `map` function on the `books` array.

Make sure to adapt the code to your specific requirements and adjust the API endpoint accordingly. Also, handle errors appropriately based on your application's needs.

9.a Course Name: Angular JS

Module Name: Server Communication using HttpClient

Create an application for Server Communication using HttpClient

Http Client Module is introduced in Angular 4 , this is a former implementation of Http Module . The HttpClient service is a part of Http Client Module only and it is used to initiate requests from server and process response with your application.

To use HttpClientService in our module , first we need to import HttpClientModule in our application. For eg:-

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

Once imported we can use HttpClient in our components , all we need to do is to inject into the class constructor. For eg:-

```
import {HttpClient} from '@angular/common/http';  
  
constructor(private http:HttpClient){ }
```

To execute request HttpClient will use XMLHttpRequest browser API to get the response. For eg:-

```
ngOnInit(){  
this.http.get('http://jsonplaceholder.com/users').subscribe(success=>{  
console.log(success);  
});  
}
```

The output will directly access the JSON response by subscribing the response which we get from the api.

Error Handling:-

A HttpRequest can fail , for various reasons , so to handle error case in our module , we have to add second callback method to subscribe. For eg:-

```
ngOnInit(){  
this.http.get('http://jsonplaceholder.com/users').subscribe(  
success=>{  
console.log(success);  
},  
error=>{  
console.log(error);  
}
```



```

}
});
}

```

To summarize, HttpClient module is used to fetch the api from service and subscribe the method and get the response in return from respective api's.

9.b Course Name: Angular JS

Module Name: Communicating with different backend services using Angular HttpClient

Create a custom service called ProductService in which Http class is used to fetch data stored in the JSON files.

Certainly! Below is an example of a `ProductService` using the `Http` class to fetch data stored in JSON files. The `Http` class is a basic wrapper around the Fetch API for making HTTP requests.

```

```jsx
// Http.js - Basic HTTP wrapper using Fetch API
class Http {
 static async get(url) {
 const response = await fetch(url);
 if (!response.ok) {
 throw new Error(`HTTP error! Status: ${response.status}`);
 }
 return await response.json();
 }
}

// ProductService.js - Custom service for fetching product data
class ProductService {
 static async getProducts() {
 try {
 // Assuming your JSON file is in the public folder, and named products.json
 const products = await Http.get('/products.json');
 return products;
 } catch (error) {
 console.error('Error fetching products:', error);
 throw error;
 }
 }
}

// Example usage in a component

```

```

import React, { useState, useEffect } from 'react';

const ProductComponent = () => {
 const [products, setProducts] = useState([]);

 useEffect(() => {
 const getProducts = async () => {
 try {
 const productsData = await ProductService.getProducts();
 setProducts(productsData);
 } catch (error) {
 // Handle error, e.g., display an error message
 }
 };

 getProducts();
 }, []);

 return (
 <div>
 <h2>Product List</h2>

 {products.map((product) => (
 <li key={product.id}>
 ID: {product.id}, Name: {product.name}

))}

 </div>
);
};

```

export default ProductComponent;

In this example:

- The `Http` class is a basic wrapper around the Fetch API, providing a simple `get` method for making GET requests.
- The `ProductService` class has a `getProducts` method that uses the `Http.get` method to fetch product data from a JSON file (assumed to be in the public folder and named `products.json`).
- The `ProductComponent` is a functional component that uses the `useState` and `useEffect` hooks to fetch and display product data.

Make sure to adjust the file paths and names based on your project structure and the location of your JSON files. Additionally, consider handling errors appropriately for your application.

## 10. a Course Name: Angular JS

### Module Name: Routing Basics, Router Links

#### Create multiple components and add routing to provide navigation between them.

The task is to enable routing between angular components by making their routes when a user clicks the link, it will be navigated to page link corresponding to the required component.

Let us know what is routing in Angular

##### Angular 8 routing:

The Angular 8 Router helps to navigate between pages that are being triggered by the user's actions. The navigation happens when the user clicks on the link or enter the URL from the browser address bar. The link can contain the reference to the router on which the user will be directed. We can also pass other parameters with a link through angular routing.

##### Approach:

- Create an Angular app.  
**syntax:**  
`ng new app_name`
- For routing you will need components, here we have made two components (home and dash) to display home page and dashboard.  
**syntax:**  
`ng g c component_name`
- In `app.module.ts`, import **RouterModule** from `@angular/router`.  
**syntax:**  
`import { RouterModule } from '@angular/router';`
- Then in imports of `app.module.ts` define the paths.
- imports: [
  - BrowserModule,
  - AppRoutingModule,
  - RouterModule.forRoot([
    - { path: 'home', component: HomeComponent },
    - { path: 'dash', component: DashComponent }
  - ])
- ],
- Now for HTML part, define the HTML for `app.component.html`. In link, define routerLink's path as the component name.
- `<a routerLink="/home">Home </a><br>`  
`<a routerLink="/dash">dashboard</a>`
- Apply router-outlet for your application in `app.component.html`. The routed views render in the `<router-outlet>`
- `<router-outlet></router-outlet>`
- Now just define HTML for `home.component.html` and `dash.component.html` file.
- Your angular web-app is ready.

##### Code Implementation:

- `app.module.ts:`

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { DashComponent } from './dash/dash.component';

@NgModule({
 declarations: [
 AppComponent,
 HomeComponent,
 DashComponent
],
 imports: [
 BrowserModule,
 AppRoutingModuleModule,
 RouterModule.forRoot([
 { path: 'home', component: HomeComponent },
 { path: 'dash', component: DashComponent }
])
],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }

```

- **app.component.html**

```

Home

dashboard
<router-outlet></router-outlet>

```

- **home.component.html**

```

<h1>GeeksforGeeks</h1>

```

- **dash.component.html**

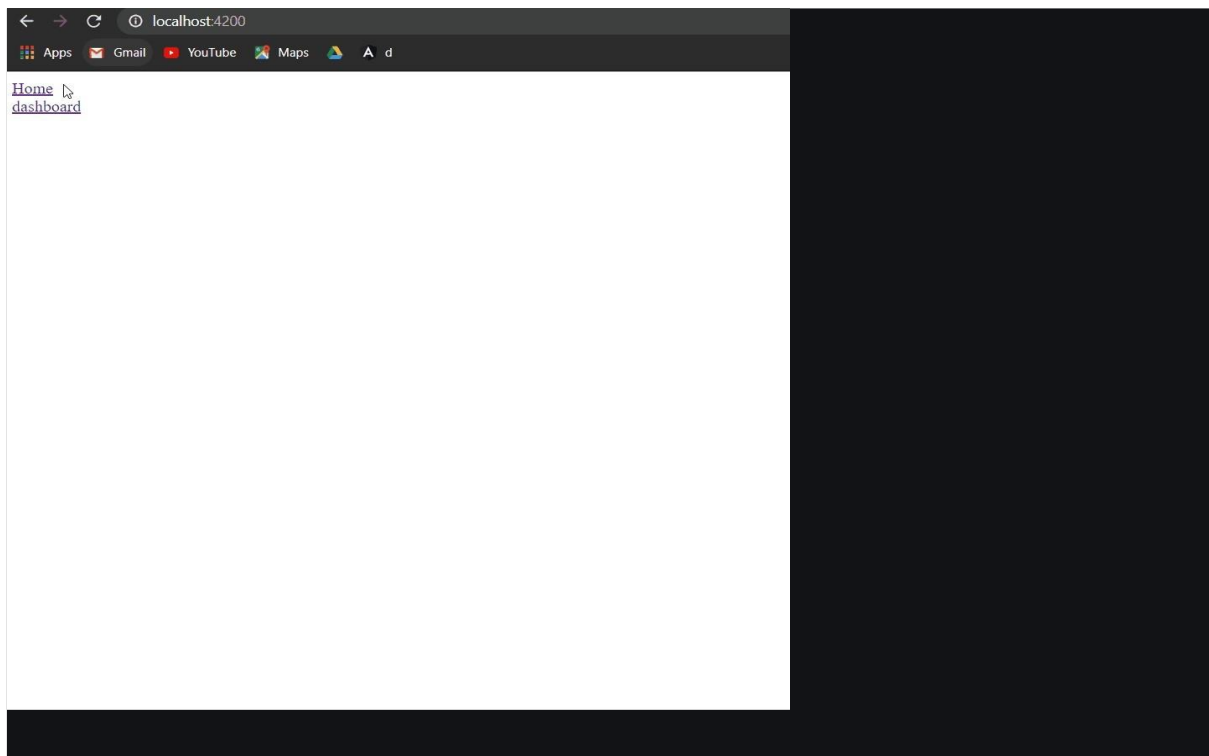
```

<h1>Hey GEEKS! Welcome to Dashboard</h1>

```

### Output:

Run the development server and click on the links:



## 10. b Course Name: Angular JS

### Module Name: Route Guards

**Considering the same example used for routing, add route guard to BooksComponent. Only after logging in, the user should be able to access BooksComponent. If the user tries to give the URL of Bookscomponent in another tab or window, or if the user tries**

To implement route guards in a React application, you can use a combination of a state management library (like Redux or React Context) and a higher-order component (HOC) for guarding routes. Below is an example of how you might implement a simple route guard using React Context and an HOC:

#### 1. **\*\*Create an AuthContext:\*\***

```
```${jsx
// AuthContext.js
import React, { createContext, useContext, useReducer } from 'react';

const AuthContext = createContext();

const initialState = {
  isAuthenticated: false,
};

const authReducer = (state, action) => {
  switch (action.type) {
    case 'LOGIN':
      return { isAuthenticated: true };
    case 'LOGOUT':
      return { isAuthenticated: false };
    default:
      return state;
  }
};
```

```

const AuthProvider = ({ children }) => {
  const [state, dispatch] = useReducer(authReducer, initialState);

  const login = () => dispatch({ type: 'LOGIN' });
  const logout = () => dispatch({ type: 'LOGOUT' });

  return (
    <AuthContext.Provider value={{ ...state, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};

const useAuth = () => {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error('useAuth must be used within an AuthProvider');
  }
  return context;
};

export { AuthProvider, useAuth };

```

2. ****Create a PrivateRoute HOC:****

```

```jsx
// PrivateRoute.js
import React from 'react';
import { Redirect, Route } from 'react-router-dom';
import { useAuth } from './AuthContext';

const PrivateRoute = ({ component: Component, ...rest }) => {
 const { isAuthenticated } = useAuth();

 return (
 <Route
 {...rest}
 render={
 (props) =>
 isAuthenticated ? (
 <Component {...props} />
) : (
 <Redirect to="/login" />
)
 }
 />
);
};

export default PrivateRoute;

```

## 3. **\*\*Modify your App component to use the AuthProvider:\*\***

```

```jsx
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import { AuthProvider } from './AuthContext';

```

```

import PrivateRoute from './PrivateRoute';
import BookComponent from './BookComponent';
import LoginComponent from './LoginComponent';

const App = () => {
  return (
    <AuthProvider>
      <Router>
        <Switch>
          <Route path="/login" component={LoginComponent} />
          <PrivateRoute path="/books" component={BookComponent} />
          { /* Other routes */ }
        </Switch>
      </Router>
    </AuthProvider>
  );
};

export default App;

```

4. ****Use the AuthContext in your LoginComponent to handle login:****

```

```jsx
// LoginComponent.js
import React from 'react';
import { useAuth } from './AuthContext';

const LoginComponent = () => {
 const { login } = useAuth();

 const handleLogin = () => {
 // Perform authentication logic (e.g., call an API)
 // If successful, call the login function
 login();
 };

 return (
 <div>
 <h2>Login</h2>
 <button onClick={handleLogin}>Login</button>
 </div>
);
};

export default LoginComponent;

```

Now, the `PrivateRoute` HOC will redirect the user to the login page if they are not authenticated when trying to access the `BookComponent`. Ensure that your login logic inside the `LoginComponent` is appropriately implemented based on your authentication requirements.

**11.a Course Name: MongoDB Essentials - A Complete MongoDB Guide**  
**Module Name: Installing MongoDB on the local computer, Create MongoDB Atlas Cluster**  
**Install MongoDB and configure ATLAS**

MongoDB has a cloud service also known as MongoDB Atlas. It is a cloud base database that allows us to host up our database and serve us whenever we need it. It removes all our effort to host our database on our local devices and can serve us anytime.

### Features of MongoDB Atlas:

- Strong security
- More precise analysis of data
- Easy Scalability
- Technical support

### Installing MongoDB Atlas:

Follow the below step to install MongoDB Atlas:

**Step 1:** Go to the [MongoDB](https://www.mongodb.com) website and signup. Enter your credentials and click on **create account** and then verify the account or just click **sign up with google** and select the Id from which you want to register.

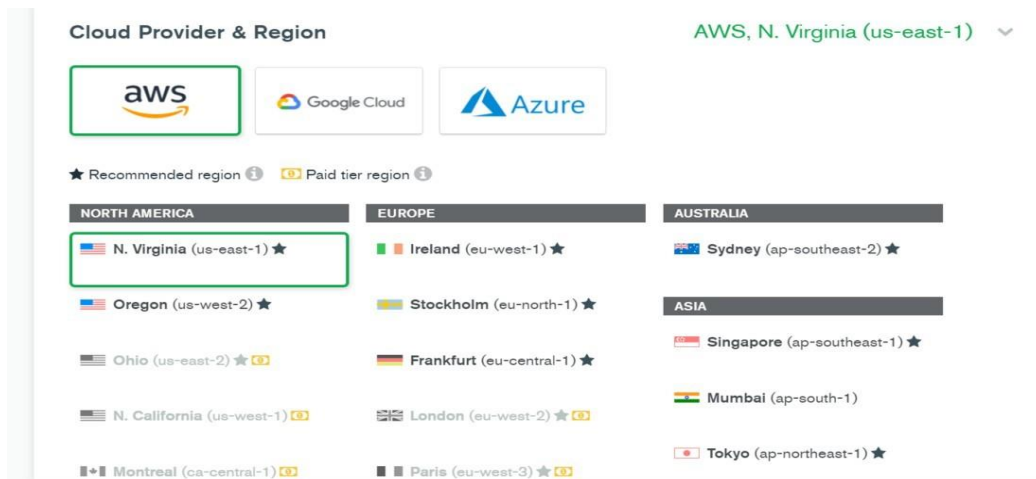
**Step 2:** Now select the **Shared option** and click on **Create** for free usage.

The screenshot displays the MongoDB Atlas pricing page with two main options: Dedicated and Shared. The Dedicated option is for production applications with advanced configuration controls, including network isolation, on-demand performance advice, and multi-region/multi-cloud options. It starts at \$0.08/hr\* (estimated cost \$56.94/month). The Shared option is for learning and exploring MongoDB in a cloud environment, offering basic configuration options, no credit card requirement, sample datasets, and an upgrade path to dedicated clusters. It is free to start. Both options have a 'Create' button.

Option	Description	Features	Cost
Dedicated	For production applications with sophisticated workload requirements. Advanced configuration controls.	✓ Network isolation and fine-grained access controls ✓ On-demand performance advice ✓ Multi-region and multi-cloud options available	Starting at \$0.08/hr* *estimated cost \$56.94/month
Shared	For learning and exploring MongoDB in a cloud environment. Basic configuration options.	✓ No credit card required to start ✓ Explore with sample datasets ✓ Upgrade to dedicated clusters for full functionality	Starting at FREE

**Step 3:** Now select cloud provider **AWS** or **Google cloud** for free hosting and then select the region in which it should be hosted. And in the Cluster tier select **M0 Sandbox** for the free environment. And If you want you can rename the cluster. It will take 4-5 minutes to set up the environment.





**Step 4:** Create a **username** and **password** then select database access to atlas Admin and network access to allow access to anywhere. That will be the username and password through which the user can access the database given to who do you give access.

✓ How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

Username and Password

Certificate

Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password. You can manage existing users via the [Database Access Page](#).

Username

Password

Enter username

Enter password

Create User

This password contains special characters which will be URL-encoded.

**Step 5:** Select **Connect** and then select **connect with the MongoDB Shell** then go to **I have MongoDB shell installed**. Select the node version from there. And after that run the command in the command line which is given. Then enter the password which the user has set earlier with the username and password.

1 Select your mongo shell version

4.4

(To check your shell version, run `mongosh --version` or `mongo --version`)

2 Run your connection string in your command line

Use this connection string in your application:

```
mongo "mongodb+srv://cluster0.████.mongodb.net/████?authSource=████" --username █████
```

Replace **myFirstDatabase** with the name of the database that connections will use by default. You will be prompted for the password for the Database User, **priyanshu**. When entering your password, make sure all special characters are URL encoded.

**Step 6: Go to **Connect** then **Connect your application** select your node version. Now copy the link given and paste it into your application where you connect MongoDB. And replace <password> with your password.**

1 Select your driver and version

DRIVER

Node.js

VERSION

4.0 or later

2 Add your connection string into your application code

☐ Include full driver code example

```
mongodb+srv://████:████<password>@cluster0.████.mongodb.net/myfirstDatabase?
```

## Exercise -12.a

**Course Name: MongoDB Essentials - A Complete MongoDB Guide**

**Module Name: Create and Delete Databases and Collections**

**Write MongoDB queries to Create and drop databases and collections.**

MongoDB has no "create" command for creating a database. Also, it is essential to note that MongoDB does not provide any specific command for creating a database. This might seem a bit agitated if you are new to this subject and database tool or in case you have used that conventional SQL as your database where you are required to create a new database, which will contain table and, you will then have to use the INSERT INTO TABLE to insert values manually within your table.

If you are eager to check the list of database that is residing with MongoDB, you can use the show dbs command. By default, it may not show your created database, and MongoDB's default database is a test.

## Show dbs

```
> show dbs
admin 0.000GB
config 0.000GB
```

In order to make a list show your database name, you have to make use of the command:

```
db.movie.insert({"name":"Avengers: Endgame"})
```

Now, when you again use the **show dbs** command, it will now show your created database name in the list.

```
config 0.000GB
> db.movie.insert({"name":"Avengers: Endgame"})
WriteResult({ "nInserted" : 1 })
> show dbs
admin 0.000GB
config 0.000GB
my_project_db 0.000GB
>
```

It is to be noted that, for checking your currently selected database, you can use the command:

```
db
```

```
my_project_db
>
```

## MongoDB delete document from a collection

- a) Using remove() method
- b) Remove only one document matching your criteria
- c) Remove all documents

## MONGODB DELETE DOCUMENTS

In MongoDB, the `db.collection.remove()` method is used to delete documents from a collection. The `remove()` method works on two parameters.

1. Deletion criteria: With the use of its syntax you can remove the documents from the collection.
2. JustOne: It removes only one document when set to true or 1.

Syntax: `db.collection_name.remove (DELETION_CRITERIA)`

### REMOVE ALL DOCUMENTS

If you want to remove all documents from a collection, pass an empty query document `{}` to the `remove()` method. The `remove()` method does not remove the indexes.

Let's take an example to demonstrate the `remove()` method. In this example, we remove all documents from the "javatpoint" collection.

```
db.javatpoint.remove({})
```

### REMOVE ALL DOCUMENTS THAT MATCH A CONDITION

If you want to remove a document that match a specific condition, call the `remove()` method with the `<query>` parameter.

The following example will remove all documents from the javatpoint collection where the type field is equal to programming language.

```
db.javatpoint.remove({ type : "programming language" })
```

### REMOVE A SINGLE DOCUMENT THAT MATCH A CONDITION

If you want to remove a single document that match a specific condition, call the `remove()` method with just One parameter set to true or 1.

The following example will remove a single document from the javatpoint collection where the type field is equal to programming language.

```
db.javatpoint.remove({ type : "programming language" }, 1)
```

**Use the Select your language drop-down menu in the upper-right to set the language of the following examples.**

By default, queries in MongoDB return all fields in matching documents. To limit the amount of data that MongoDB sends to applications, you can include a projection document to specify or restrict fields to return

This page provides examples of query operations with projection using the `db.collection.find()` method in `mongosh`. The examples on this page use the inventory collection. To populate the inventory collection, run the following:

### Return All Fields in Matching Documents

If you do not specify a projection document, the `db.collection.find()` method returns all fields in the matching documents

The following example returns all fields from all documents in the inventory collection where the status equals "A":

```
db.inventory.find({ status: "A" })
```

MongoDB Shell

The operation corresponds to the following SQL statement:

```
SELECT * from inventory WHERE status = "A"
```

Return the Specified Fields and the `_id` Field Only A projection can explicitly include several fields by setting the to 1 in the projection document. The following operation returns all documents that match the query. In the result set, only the item, status and, by default, the `_id` fields return in the matching documents.

```
db.inventory.find({ status: "A" }, { item: 1, status: 1 })
```

The operation corresponds to the following SQL statement:

```
SELECT _id, item, status from inventory WHERE status = "A"
```

Suppress `_id` Field

You can remove the `_id` field from the results by setting it to 0 in the projection, as in the following example:

```
db.inventory.find({ status: "A" }, { item: 1, status: 1, _id: 0 })
```

MongoDB Shell

The operation corresponds to the following SQL statement:

```
SELECT item, status from inventory
WHERE status = "A"
```

## Exercise -12.b

## Course Name: MongoDB Essentials – A Complete MongoDB Guide

### Module Name:

#### Introduction to MongoDB Queries

Write MongoDB queries to work with records using `find()`, `limit()`, `sort()`, `createIndex()`, `aggregate()`

#### The Limit() Method

To limit the records in MongoDB, you need to use `limit()` method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

#### Syntax

The basic syntax of `limit()` method is as follows –

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

#### Example

Consider the collection `myycol` has the following data.

```
{_id:ObjectId("507f191e810c19729de860e1"),title:"MongoDBOverview"},
{_id:ObjectId("507f191e810c19729de860e2"),
title:"NoSQLOverview"},
{_id:ObjectId("507f191e810c19729de860e3"),title:"TutorialsPointOverview"}
```

Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDBOverview"}
{"title":"NoSQLOverview"}
>
```

If you don't specify the number argument in `limit()` method then it will display all documents from the collection.