PS03CMCA51 Web Application Frameworks (Angular Part)

Dr. J. V. Smart

Table of Contents

- Angular Pipes
    - RxJS Operators
- Angular Routing
- RESTful Web Services
- CORS Policy
- JavaScript Notes
- TypeScript Introduction and Reference
- Additional Notes

Syllabus

Syllabus with effect from the Academic Year 2021-2022

Course Code	PS03CMCA51	Title of the Course	WEB APPLICATION FRAMEWORKS
Total Credits of the Course	4	Hours per Week	4

Course Objectives:	1. To learn the fundamentals of the Python programming language. 2. To study development of procedural as well as object-oriented Python programs. 3. To learn GUI program development using Python. 4. To understand how to access files and databases from Python. 5. To learn client-side web application frameworks. 6. To learn server-side web application frameworks.

Course Content

Unit	Description	Weightage* (%)
1.	**Basic Web Application Development Tools** - Introduction to HTML5, CSS3 - Interactive web pages using JavaScript - The JQuery library - JavaScript user interface library	25

Unit	Description	Weightage* (%)
2.	**Web Frameworks for Python** - Introduction to web frameworks - Popular full-stack frameworks and non full-stack frameworks (microframeworks) - Working with Flask and Django frameworks.	25
3.	**Client-side Web Application Frameworks** - Setting up Project, project organization and management - Templates - MVC Architecture - Data binding - Dependency injection - Routing	25
4.	**Server-side Web Application frameworks** - Application structure - MVC Architecture - Routing - Helpers - Libraries - Form validation - Session management - Active record	25

Teaching-Learning Methodology	Blended learning approach incorporating traditional classroom teaching as well as online / ICT-based teaching practices

Evaluation Pattern

Sr. No.	Details of the Evaluation	Weightage
1.	Internal Written / Practical Examination (As per CBCS R.6.8.3)	15%

Sr. No.	Details of the Evaluation	Weightage
2.	Internal Continuous Assessment in the form of Practical, Viva-voce, Quizzes, Seminars, Assignments, Attendance (As per CBCS R.6.8.3)	15%
3.	University Examination	70%

Course Outcomes: Having completed this course, the learner will be able to	
1.	develop websites using Django Framework.
2.	manipulate different Python data types.
3.	develop object-oriented programs using Python.
4.	understand the Python package system.
5.	create basic GUI programs as well as Python programs with file handling and database access.

Suggested References:

Sr. No.	References
1.	Dane Cameron, “HTML5, JavaScript and jQuery”, Wrox publication.
2.	David Sawyer McFarland, “CSS3”, O’reilly.
3.	Brad Green and Syham Seshadri, “AngularJS”, O’Reilly.
4.	Python Web Frameworks by Carlos de la Guardia, O'Reilly Media, Inc., March 2016.
5.	Jake Spurlock, “Bootstrap”, O’Reilly.
6.	Thomas Myer, “Professional CodeIgniter”, Wrox Professional Guides.
7.	Karl Swedberg, Jonathan Chaffer, “jQuery 1.4 Reference Guide”, PACKT publishing.
8.	Valeri Karpov, Diego Netto, “Professional AngularJS”, Wrox publication.

Sr. No.	References
9.	Zak Ruvalcaba, Anne Boehm, “HTML5 and CSS3”, Murach.
10.	Bear Bibeault, Yehuda Katz, “jQuery in action”, 2nd edition, Dreamtech press.

On-line resources to be used if available as reference material	
1.	Python documentation.

Glossary

WWW
World-Wide Web

MVC
Model-View-Controller

MVVM
Model-View-ViewModel

Node.js (Node)
An open source JavaScript runtime for developing server-side and desktop applications, libraries, packages and APIs using JavaScript

Visual Studio Code (VS Code)
Visual Studio Code (VS Code) is a cross-platform open source code editor from Microsoft.

npm
Node Package Manager

Angular CLI
Angular Commandline Interface

ng
A**ng**ular

Data Binding
Data binding is a technique for connecting two representations of a piece of data such that a change is one source of data is reflected in the other.

Dependency Injection

Dependency injection is a technique whereby one object (the injector) provides the dependencies (injectables) of another object (the client) at runtime.

Asynchronous Execution

When a request / call is made to a function / method / service and the caller does not wait for the response from the service / return from the call; such mode of execution is called asynchronous execution.

RxJS

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

Observables

An observable is an RxJS construct for making asynchronous requests.

Subscription

A subscription fetches the value(s) from an observale and returns them.

HTTP

HyperText Transfer Protocol

XML

eXtensible Markup Language

WSDL

Web Services Description Language

UDDI

Universal Description, Discovery, and Integration

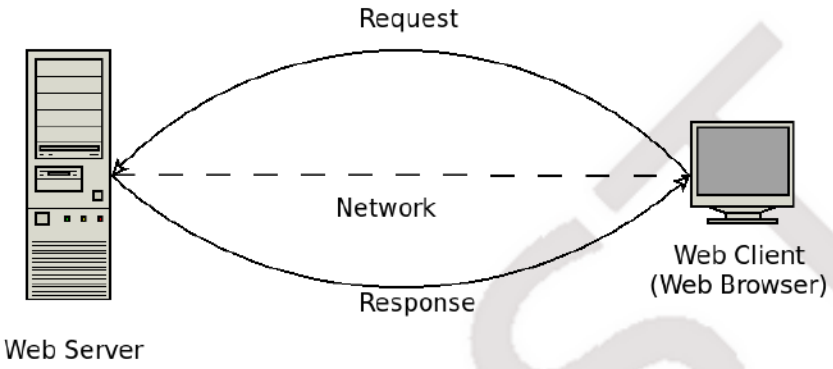
JSON

JavaScript Object Notation

REST

REpresentational State Transfer

The World Wide Web Architecture



The World Wide Web Architecture

Introduction to the Angular Client-side Web Application Framework

Angular (earlier called AngularJS) is an open-source client-side web application framework developed by Google. The first version of the framework used the JavaScript programming language and was called *AngularJS*. The second version was a major rewrite of the whole framework. It switched to TypeScript, a superset of JavaScript. Starting with version 2, the framework is called simply *Angular*. The latest version of the Angular framework is 14.

The official Angular website is https://angular.io/

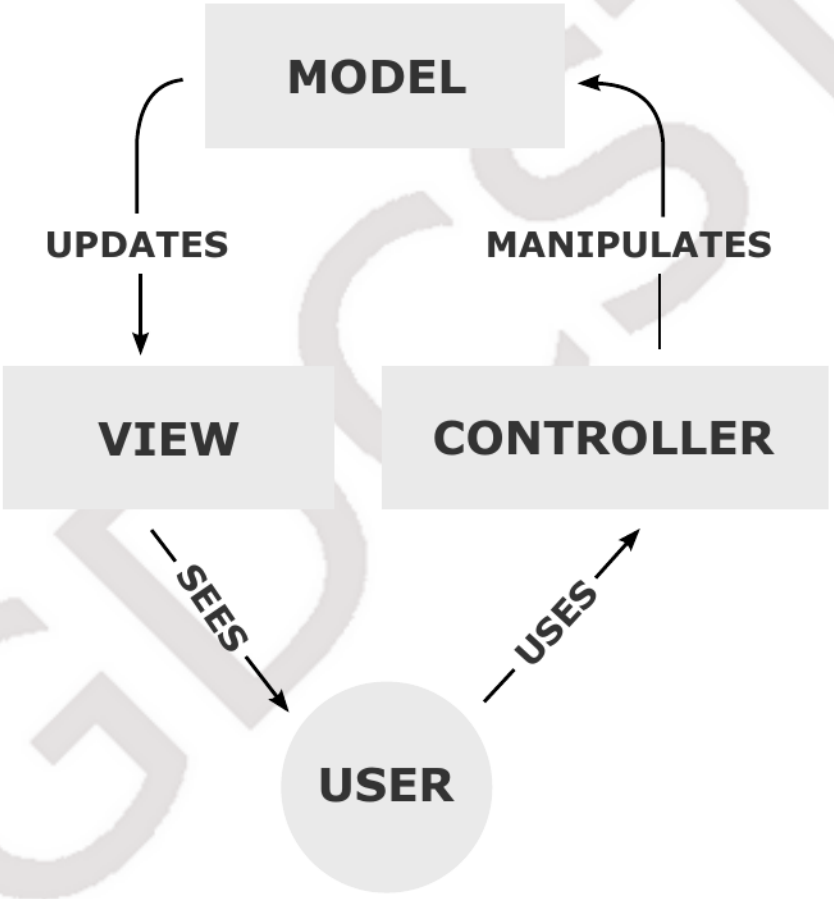
The Angular Logo

The TypeScript Programming Language

TypeScript is a programming language developed and maintained by Microsoft. It is a superset of JavaScript. It adds optional static typing to the language. One can optionally specify the data type of a variable using the syntax `variable : datatype`. TypeScript programs are usually compiled to JavaScript. Extension of TypeScript files is `.ts`.

MVC (Model-View-Controller)

MVC (Model-View-Controller) is an architectural pattern for software design. As the name suggets, an MVC application consists of three parts - Model, View and Controller.



The MVC Pattern

- **Model** The model is responsible for storing and manipulating the data of the application. It is also responsible for implementing the business logic
- **View** The view is a representation of (part of) the data that is presented to the user. Same data may be presented in different form by different views (e.g. table v/s chart)
- **Controller** The controller accepts user input in the form of events,

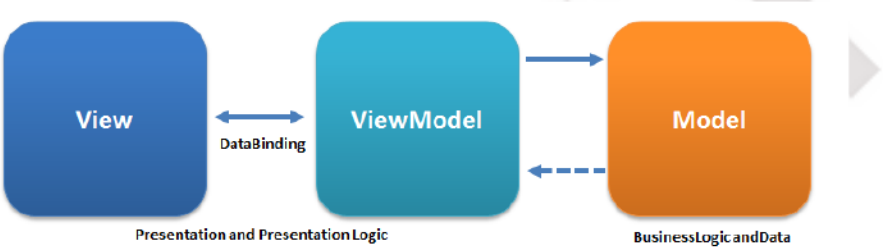
optionally validate it and then either sends it to the model for action or selects the appropriate view to be displayed. Often, it is also responsible for fetching the data needed by the view from the model

Major advantages of the MVC pattern include separation between presentation and logic, possibility of simultaneous independent development of the model, view and controller components and the ability to have multiple views for presenting the same data in different ways. MVC pattern was conceived for GUI application development, but it is heavily used for web application development. Different frameworks implement the pattern in different ways. Several software architectural patterns based on MVC have been developed.

AngularJS (the first version of Angular) used MVC. Angular (version 2 onward) follows a component architecture and broadly follows the MVVM pattern.

MVVM (Model-View-ViewModel)

MVVM (Model-View-ViewModel) is an architectural pattern for software design that evolved from the MVC pattern. An MVVM application consists of three parts - Model, View and ViewModel. In most cases, MVVM implementations have data binding between the view and the view model.



The MVVM Pattern

- **Model** The model is responsible for storing and manipulating the data of the application. It is also responsible for implementing the business logic
- **View** The view is a representation of (part of) the data that is presented to the user. Same data may be presented in different form by different views (e.g. table v/s chart). The view gets the data to be displayed from the view model and presents them to the user. The view is also responsible for accepting user input, potentially validating it and then updating the view model. The view contains only the UX (User eXperience) elements and is developed by the designers. It contains almost no code
- **View Model** The view model fetches from the model the part of the data to be presented by the view. This component contains the code for fetching

the data, but has no user interface elements. The view model exposes the data from the model to the view, which presents them to the user

- **Data Binding** In most MVVM implementations, there is a two-way binding between the display elements in the view and the data in the view model; such that whenever the data in the view changes, the view model is updated automatically, and vice versa

A major advantage of the MVVM pattern over the MVC pattern is the complete separation between the UX (user interface design) and the coding needed to fetch the data to be displayed and to propagate the modifications made by the user in the view to the model. Another important advantage of the MVVM model is that since the views contain no code and the view model contains a representation of the view, automated testing frameworks can easily test the application by simulating user actions in the view by calling the methods in the view model that handle those actions.

The disadvantage of the MVVM pattern is the additional memory needed to hold the view model and the performance impact of automatic data binding.

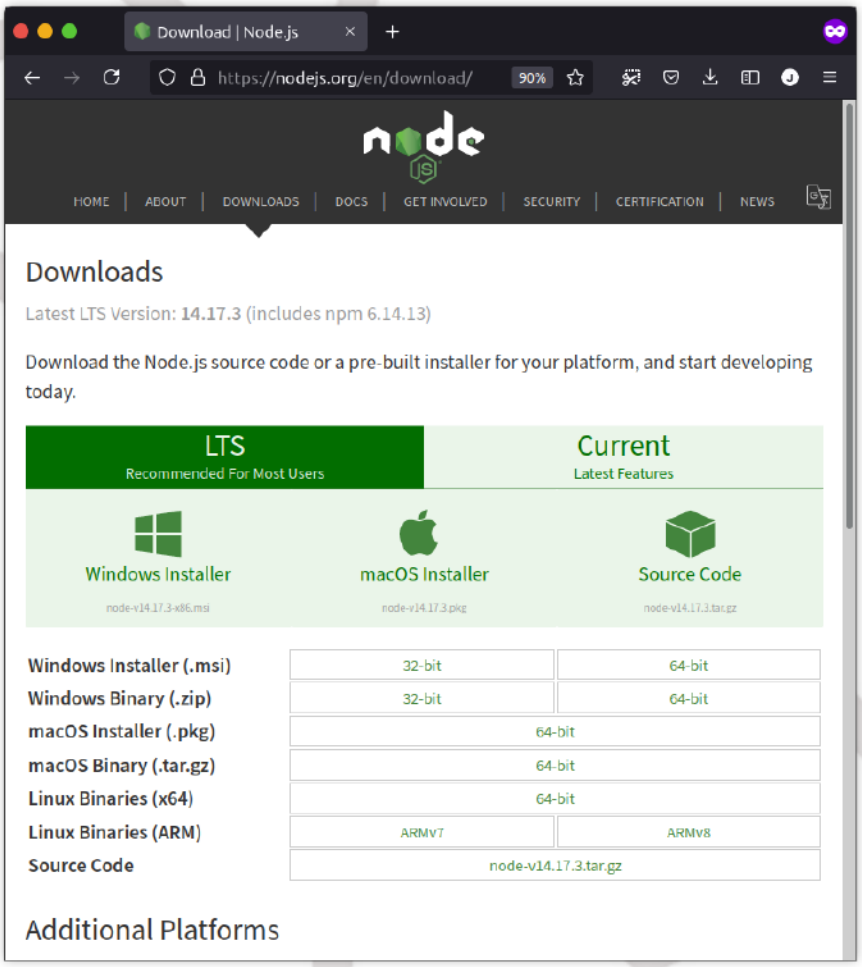
In Angular, the model can be an object or array of objects of a TypeScript class that is like a POJO class (Plain Old Java Object). The HTML and CSS parts of the component represent the view. The TypeScript part of the component corresponds to the view model.

Installation

Angular is developed using Node.js. Node.js is a free and open-source framework that allows one to use the JavaScript programming language for developing the server-side part of a web application as well as desktop applications and command line applications. Thousands of packages have been developed using Node.js. These plugin packages can be installed using the `npm` (node package manager) command.

Installing Node.js

Download and install Node.js from its official website using the URL https://nodejs.org/en/download/.



Downloading Node.js

Installing Angular

The Angular framework is a Node.js package and can be installed using `npm`.

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

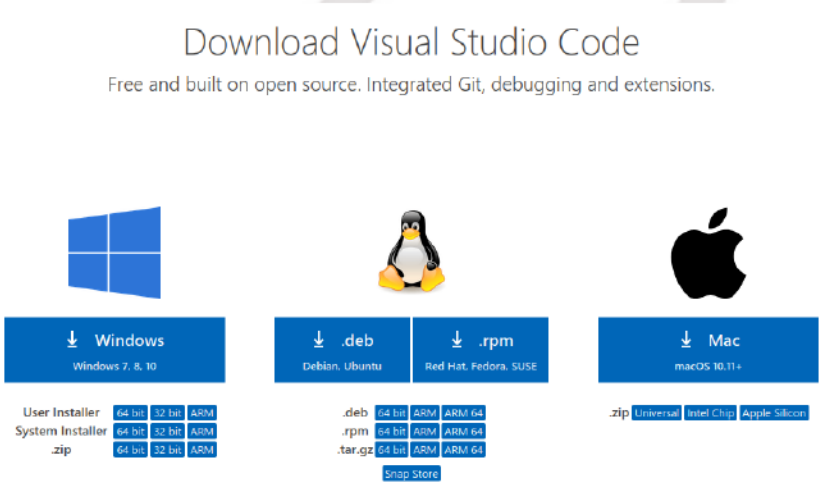
Once Angular is installed, one may use the Angular CLI (Command Line Interface) to create and manage Angular applications.

Installing Visual Studio Code (VS Code)

Visual Studio Code is a cross-platform open source code editor from Microsoft (it is different from the Visual Studio IDE). It is available for Windows, macOS and Linux.

Download VS Code from its official website https://code.visualstudio.com. For Windows, the *User Installer* will install VS Code for the current user only. The *System Installer* will install VS Code for all users, but requires administrative privileges. The *zip* download is a portable package that does not require

installation - you just unzip it and you are ready to go.



Download VS Code

VS Code supports only some tools and technologies out-of-box. However, it has a plugin architecture. A large number of extensions are available in the market place that add support for various tools, technologies and features.

In VS Code, there is a large number of commands. Also, various extensions also add their own commands. It is not possible to accommodate all the commands in the menu system. Hence many commands are executed by searching in the *command palette* or using shortcut keys. The command palette can be opened by using the shortcut key `CTRL+SHIFT+P`.

To add support for Angular Development, run the *extensions: Install Extensions* command (or select the extensions icon from the left-hand side side bar) and install the "Angular Extension Pack" extension. This pack will install several Angular-related extensions.

Creating an Angular App

An Angular app cannot be created from VS Code. It must be created from the Angular CLI (command line interface).

Open a terminal (in Windows, press `Windows+R cmd ENTER`). Go to the directory where you want to create the app. Type the following command:

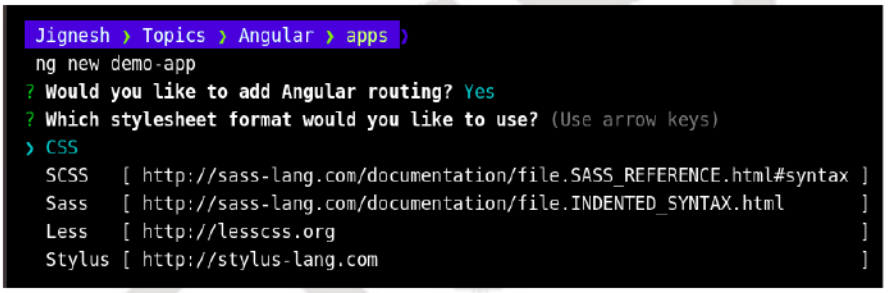
```
ng new <app_name>
```

Note: When you use this command for the first time, you may have to provide your GitHub account details. Create a free account if you don't have one and

then type the following commands:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

The Angular CLI will prompt for whether you want to include Angular routing or not and which stylesheet format do you want to use (use arrow keys to select the format). After that, it will create the app. This may take some time.



Creating Angular App

This will create the basic project structure in a folder named `<app_name>`. The subfolder src → app contains the source code. It creates a root module in the file `app.module.ts`, It also creates a root component. If we chose to include support for Angular routing, it creates the file `app-routing.module.ts`.

Open VS Code. If any workspace is open, close it. Select `File` → `Add Folder to Workspace…` and select the folder containing the app. The workspace can be saved by selecting the option `File` → `Save Workspace As…` and providing a file name.

Open a terminal in VS Code by selecting `Terminal` → `New Terminal`.

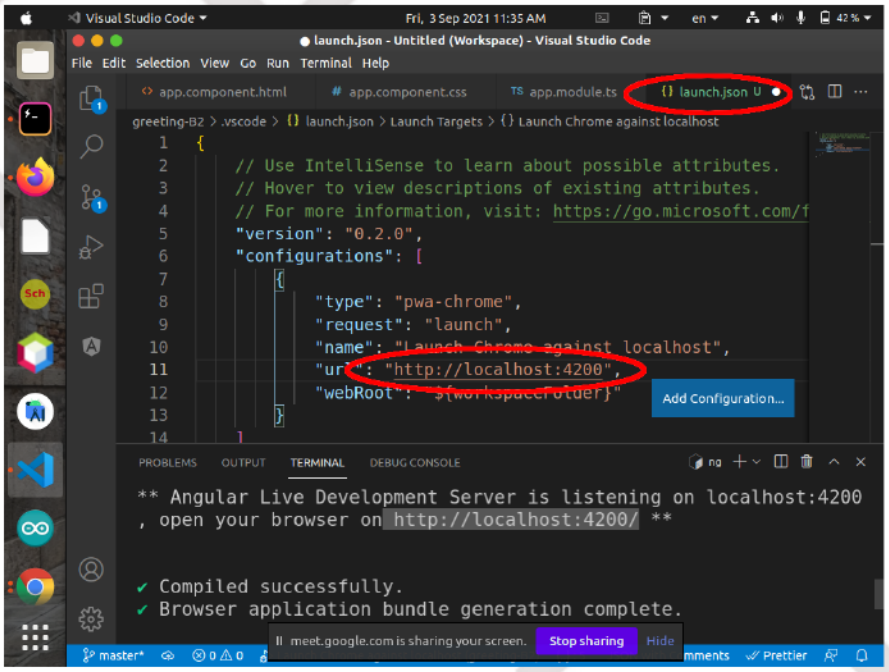
You can start the development web server to serve the app by typing the command

```
ng serve
```

in the terminal. You can open the app in the system's default browser by appending the option `--open` to the command. However, that will not attach the debug console of VS Code with the debug console of the browser.

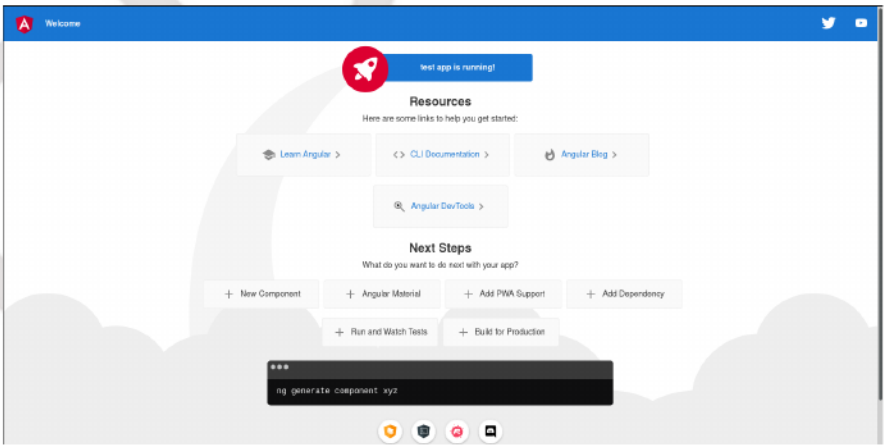
To run the app, open it in the Google Chrome browser and attach the debug console of VS Code to the browser's debug console; select `Run` → `Start Debugging`. When you do this for the first time, it opens a file `launch.json`. You need to change the port number in the url from `8080` to `4200` and save.

After this, you can start the app using the `Run` → `Start Debugging`.



Launch Configuration

By default, Angular CLI creates the root component containing links to some Angular related resources. To remove this content, delete everything from the HTML template of the root element (`app.component.html`). A `title` property is also defined in the component class (`app.component.ts`). It can be modified if desired. The title property of the component class can be accessed in the HTML template using string interpolation like this: `{{title}}` . Styling for the root component can be specified in `app.component.css` .



Default App

Data Binding

Data binding is a technique for connecting two representations of a piece of data such that a change is one source of data is reflected in the other. Data binding

can be one-way or two-way. In one-way data binding, the changes in one representation of data are reflected in the other, but the converse (opposite) is not true. In two way data binding, changes in any representation of data are reflected in the other representation.

Types of Data Binding in Angular

1. **String Interpolation** This is a type of one-way data binding where a string contains some code (called template expression) enclosed in a set of double curly braces (`{{ code }}`). As the templates are reevaluated after every event; for performance reasons, there are many restrictions on the type of code that is allowed in a template expression. The common use is to get the value of a component variable / component property using `{{ variable }}` or `{{ object.member }}` . This is a one-way binding from the view model to the view; meaning that only changes to the view model are propagated to the view, but changes to the view are not propagated to the view model
2. **Property Binding** Property Binding allows us to bind some property of a view element to a template expression. The property will change automatically when the value of the template expression changes. This is a one-way binding from the view model to the view; meaning that only changes to the view model are propagated to the view, but changes to the view are not propagated to the view model. The syntax of property binding uses square brackets: `[property] = "template_expression"` . For example, `[disabled]="isDisabled"` where `isDisabled: booolean` is a variable in the component class.
3. **Event Binding** Event binding binds some event of a view element to code in the view model. This is a one-way binding from the view to the view model; meaning that an event in the view can update the view model, but a change in the view model will not affect the view. The syntax of event binding uses round brackets: `(event)="template_expression"` . For example, `(click)='reloadBook();'` where `reloadBook()` is a method in the component class.
4. **Two-Way Data Binding** Two-way data binding is a combination of both property and event binding and it is a continuous synchronization of data between the view and the view model, which means that any changes to the view are propagated to the view model and any changes to the view model are propagated to the view immediately. Its syntax combines the syntax for property binding and event binding - it uses round brackets inside square brackets: `[(ngModel)]="book.title"` . To be able to use

`ngModel`, one must add the `FormsModule` to the `imports` array in `app.module.ts`. `FormsModule` must also be imported at the top in the component TypeScript file.

Angular Components

An Angular component represents a visual element on the web page. An Angular component usually has its own directory. An Angular component has four files.

- **.ts** This file contains the TypeScript code. It defines a component class that represents the view model
- **.html** This file contains the HTML template that represents the view
- **.css** This file may contain styling specific to this particular component
- **.spec.ts** this file contains code for unit testing of the component

An Angular component can be created by using the command

```
ng create component <component-name>
```

in the Angular CLI. It can also be created by right-clicking the app subfolder in the VS Code Explorer, selecting the option `Generate Component` and typing in the name of the component.

An Angular class becomes a component when decorated with the `@Component` decorator.

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

The selector specification creates a custom HTML element tag by that name (`<app-employees>` in the above example).

Angular Services

An Angular service provides a specific functionality. Unlike a component, a service does not have any user interface. A service runs in the background asynchronously. It can be used anywhere in the app via dependency injection. An Angular class becomes a service when it is decorated with the `@Injectable` decorator. `providedIn: 'root'` indicates that the service can be injected in the root component and any of its child components, i.e. everywhere in the app.

```
@Injectable({
  providedIn: 'root'
})
export class EmployeeService {
    ...
}
```

An Angular service can be created by using the command

```
ng create service <service-name>
```

in the Angular CLI. It can also be created by right-clicking the app subfolder in the VS Code Explorer, selecting the option `Generate Service` and typing in the name of the service.

Imports and Exports in Angular

In Angular, a definition (class, interface, component, service, etc.) is only available in the same program file by default. To make it available elsewhere in the app, one must use the keyword `export` in front of it.

A definition must be imported in a program file before it can be used in it. User-defined classes, interfaces, etc. can be imported by using the following syntax:

```
import { MessageService } from '../Message.service';
```

Here, we specify the name of the definition to be imported in curly braces (`{ MessageService}`) and the relative path of the file from which it is to be imported. In the relative path, `./` is used for a file in the current directory, while `../` is used for a file in the parent directory. The file extension `.ts` is omitted as it is assumed by default.

Built-in definitions from the Angular API can be imported as per the following examples:

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

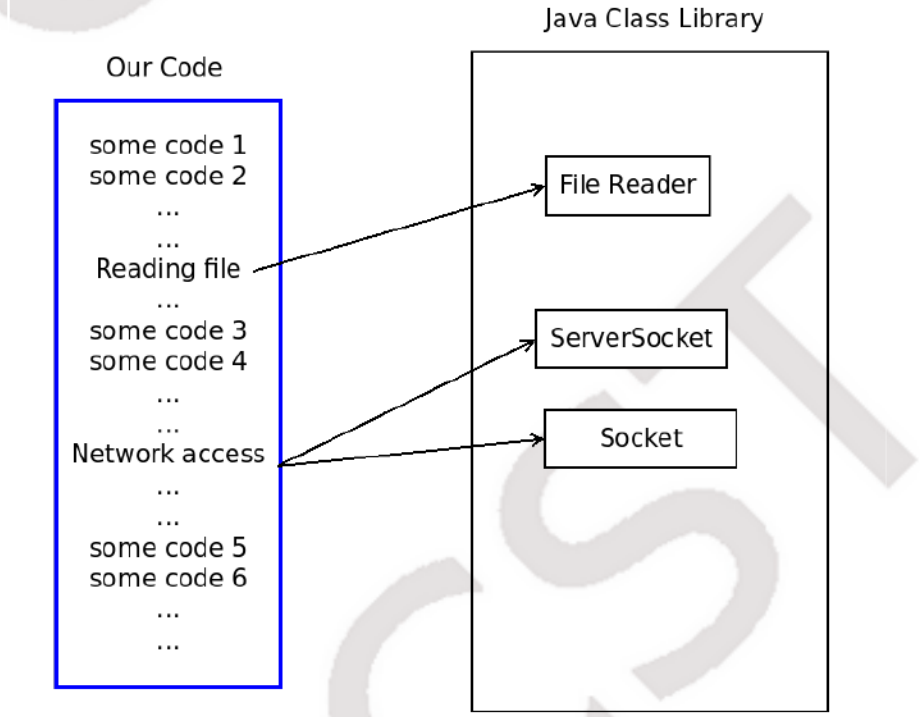
All the modules, components and services that are to be used throughout the app must be added to the `imports` array in the root module (`app.module.ts`) as follows. Before one can write a definition in the `imports` array, it must be imported in the `app.module.ts` file. However, if we type a name in the `imports` array in VS Code, it adds an import at the top automatically. When we create a component or service using an `ng generate` command or the context menu item in VS Code, its entry is added to the `imports` array in `app.module.ts` automatically.

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { EmployeesComponent } from './employees
        /employees.component';
import { HttpClientModule } from '@angular/common/http';
import { MessageComponent } from './message/message.component';
import { EmployeeDetailComponent } from './EmployeeDetail
        /EmployeeDetail.component';
import { EmployeeAddComponent } from './employee-add/employee-
        add.component';
import { FormsModule } from '@angular/forms';
import { EmployeeUpdateComponent } from './employee-update
        /employee-update.component';
import { DeleteEmployeeComponent } from './delete-employee
        /delete-employee.component';

@NgModule({
  declarations: [
    AppComponent,
      EmployeesComponent,
      MessageComponent,
      EmployeeDetailComponent,
      EmployeeAddComponent,
      EmployeeUpdateComponent,
      DeleteEmployeeComponent
   ],
  imports: [
    BrowserModule,
```

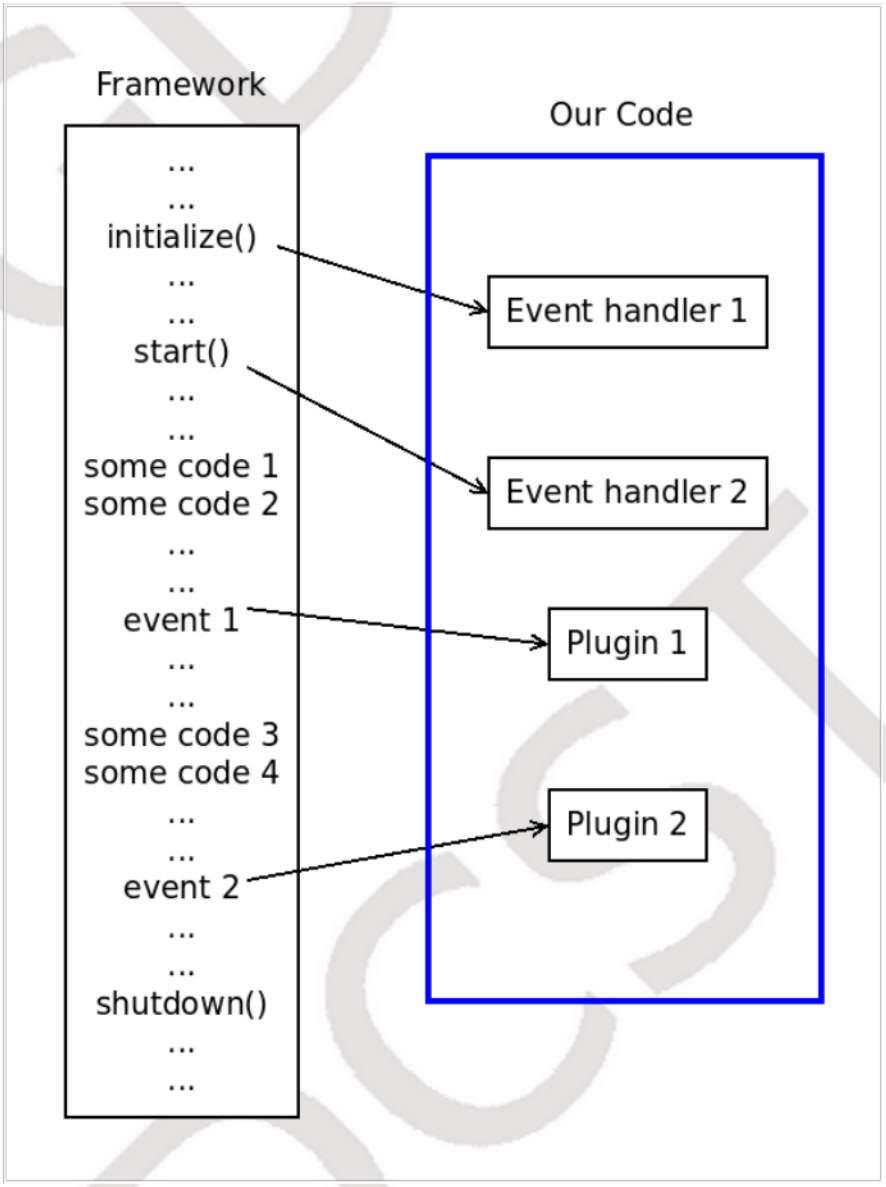
```
    AppRoutingModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Inversion of Control (IoC)

IoC is a programming methodology in software engineering. IoC inverts the flow of control as compared to traditional programming. In traditional programming, a programmer writes custom code for specific requirements that calls functions or methods from readymade generic reusable libraries for common functionality. In IoC, there is a readymade generic framework that calls the programmer-written custom code for specific requirements. The generic framework, called IoC container, provides common functionality and has a plugin architecture in which the programmer can plug or attach custom code as per the specific requirements. Dependency injection is an example of a software design pattern that follows inversion of control.



Traditional Flow of Control



Inverted Flow of Control

Dependency Injection

In software engineering, dependency injection is a technique in which an object receives other objects that it depends on. These other objects are called dependencies. In code that does not use dependency injection, the dependencies have to be hard-coded, making it difficult to change them. With dependency injection, the dependency is "injected", i.e. passed, to the object via the constructor, a setter method or an interface that is implemented by the class. This makes it easy to substitute one dependency with another without affecting the client code. In software engineering terms, dependency injection reduces coupling between code.

When one object requires the services / functionality provided by another object, the first object is called the client object / client code and the second object is

called its dependency. Dependency injection is a technique whereby one object (the injector) provides the dependencies (injectables) of another object (the client) at runtime. A dependency is any object required by the client to perform its task. The dependency object is said to be injected into the client. Typically, a dependency provides some kind of service. Hence it is also called service. In Angular, a dependency is called `Injectable`. Dependency injection is one of the techniques for implementing the *Inversion of Control (IoC)* design pattern. Usually, the dependency injection is carried out automatically by a framework. In case of Angular, the injector is provided by the Angular framework.

Advantages

- **Separation of Concern and Reduction in Coupling** Each *concern* (functionality) in a software project is handled by a separate component or module, increasing the modularity and reducing coupling
- **Increased Configurability** The services actually used by the client can be configured separately, often via a configuration file or code
- **Ease of Unit Testing** Client code can be subjected to unit testing easily by using *mock* or *stub* implementations of the dependencies (services) instead of the real ones
- **Simultaneous Independent Development** The client code and the services can be developed and unit-tested independently and simultaneously by different teams

Main Roles / Components

- **Client**
- **Service (dependency / injectable)**
- **Service Interface**
- **Injector**

Examples

```
// Without dependency injection
export class BookGetComponent implements OnInit {

  books: Book[];
  bs : BookService;

  constructor(private bs: BookService) {
```

```
// The service class BookService is hard-coded and cannot be
        replaced by another
// class providing the same service

    bs = new BookService();

}

  ngOnInit() {
    this.bs
      .getBooks()
      .subscribe((data: Book[]) => {
        this.books = data;
    });
  }
}
```

```
// With dependency injection via the constructor
export class BookGetComponent implements OnInit {

  books: Book[];

// The BookService object bs is "injected" (provided) via the
        cosntructor
// When calling the cosntructor, it can be replaced by any other
        object
// that implements the same interface

  constructor(private bs: BookService) { }

  ngOnInit() {
    this.bs
      .getBooks()
      .subscribe((data: Book[]) => {
        this.books = data;
    });
  }
}
```

```
// With dependency injection via a setter method
export class BookGetComponent implements OnInit {

  books: Book[];
  bs : BookService;

// The BookService object bs is "injected" (provided) by the
        setter method
// When calling the setter method, it can be replaced by any
        other object
// that implements the same interface
```

```
    setBookService(private newBS: BookService) {
    this.bs = newBS;
  }

  constructor() { }

  ngOnInit() {
    this.bs
      .getBooks()
      .subscribe((data: Book[]) => {
        this.books = data;
    });
  }
}
```

Dependency Injection in Angular

In Angular, dependency injection is typically done by passing a parameter to the constructor. By default, the parameter is local to the constructor and not available anywhere else in the class. However, if we use any access modifier (`public` , `private` or `protected`) or the keyword `readonly` in front of the parameter, then Angular automatically declares a property having the same name in the class and assigns the parameter value to that property. In this way, it can be accessed from anywhere within the class by using the `this.service` syntax. Usually, we use the access modifier `private` .

```
  constructor(private employeeService: EmployeeService) { }
```

will inject an object of `EmployeeService` called `employeeService` into the class. It will also define a property with the name `employeeService` in the class and assign the parameter to that property. Hence, the above code is equivalent to

```
  employeeService: EmployeeService;

  constructor(employeeService: EmployeeService) {
    this.employeeService = employeeService;
}
```

Asynchronous Execution

When a request / call is made to a function / method / service and the caller does not wait for the response from the service / return from the call; such mode of execution is called asynchronous execution.

Synchronous (blocking) v/s asynchronous (non-blocking) operations:

```
// Synchronous operation concept
String fn1(int p1, float p2i) {
    ...
    ...
    return str;
}

String s1 = null;
main() {
    ...
    s1 = fn1(5, 7,2);
    System.out.println("Success" + s1);
    // process the string returned by fn1()...
    ...
}
```

```
// Asynchronous operation concept
void fn1(int p1, float p2i, function sucess, function fail) {
    ...
    ...
    // call success() with return value of the function (String)
        as a parameter;
}

// Callback function to be used when fn1() succeeds
void success(String returnValue) {
    // process the string returned by fn1()...
}

// Callback function to be used when fn1() fails
void fail(int errCode, String errMsg) {
    ...
}

main() {
    ...
    fn1(5, 7,2, success, fail);
    System.out.println("Next statement...");
    ...
}
```

```
      Web Server
        <------------------------------- Web Client (Web
Browser)
        |      request                          ^
        |                                       |
        |               Network                 |
        |                                       |
        |---------------------------------------
                response
```

Asynchronous requests (AJAX):

- XMLHttpRequest objects
- JavaScript Promises
- RxJS and Observables

Angular Directives

The *ngIf directive

The `*ngIf` directive is a conditional directive in Angular. It is used as an attribute of an HTML element in the HTML template of the view. It accepts a boolean expression as an argument. If the boolean expression is true, that element along with its child elements, if any, is created. If the boolean expression is false; the element, including child element(s), if any, is not created. The boolean expression is evaluated as per *truthiness* rules.

```
<h3>Employee Details</h3>
<div *ngIf="employee">
<table>
  <tr>
  <td class="label-td"><label for="employeeNumber">Employee
        Number</label></td>
  <td><span id="employeeNumber">{{employee.employeeNumber}}
        </span></td>
</tr>
<tr>
  <td class="label-td"><label for="lastName">Last
        Name</label></td>
  <td><span id="lastName">{{employee.lastName}}</span></td>
</tr>
    ...
    ...
</table>
```

```
<div>
```

The *ngFor directive

The `*ngFor` directive is the *repeater direcrtive* in Angular. It is used as an attribute of an HTML element in the HTML template of the view. The directive accepts a looping expression of the form `let <element> of <list>`. The directive iterates over the `<list>` and, in each iteration, assigns one value from the list to `<element>`. The HTML element is repeatedly created for each iteration, with corresponding value of `<element>`.

```
// Angular
*ngFor="let hero of heroes"
```

```
// JavaScript
for (var hero in heroes) {

}
```

```
// Java
for (Hero hero : heroes) {

}
```

```
// C#
foreach(Hero hero in heroes) {

}
```

Classes and Interfaces in TypeScript

Classes and interfaces are powerful structures that facilitate not just object-oriented programming but also type-checking in TypeScript. A class is a blueprint from which we can create objects that share the same configuration - properties and methods. An interface is a group of related properties and methods that describe an object, but neither provides implementation nor initialisation for them.

There is a difference in the syntax for classes and interfaces. Interface

properties can end in commas or semi-colons, however class properties can only end in semi-colons.

Interfaces can be *implemented* or *extended* but cannot be instantiated (the *new* operator cannot be used on them). They get removed when transpiling (compiling) to JavaScript.

When not to use interfaces: When you want to have default values, implementations, constructors, or function definitions (not just signatures).

When to use classes: When you want to create objects that have actual function code in them, have a constructor for initialization, and/or you want to create instances of them with the *new* operator.

TypeScript Interfaces Disappear During Compilation

TypeScript supports interfaces, but JavaScript does not. Hence, interfaces defined in TypeScript disappear when the TypeScript code is compiled to JavaScript.

```
var l1: number[] = [ 10, 20, 30, 40 ];

for (var ele in l1) {
    console.log(l1[ele]);
}

var obj1 = {
    no: 10,
    name: "abc",
};

interface Student {
    id : number;
    name: string;
}

var obj2:Student = {
    id: 101,
    name: 'Xyz'
}

console.log(obj2);
```

```
tsc test.ts
```

```
var l1 = [10, 20, 30, 40];
for (var ele in l1) {
    console.log(l1[ele]);
}
var obj1 = {
    no: 10,
    name: "abc"
};
var obj2 = {
    id: 101,
    name: 'Xyz'
};
console.log(obj2);
/////// OUTPUT ///////
10
20
30
40
{ id: 101, name: 'Xyz' }
```

RxJS

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

Observables

An `Observable` is an RxJS construct for making asynchronous requests. An observable makes a request and returns data as it receives them. An observable can be *observed* or *monitored* for the arrival of new data by subscribing to it. Whenever the observable receives data, it notifies the subscriber.

An observable is evaluated lazily. That means that it does not make any requests until there is at least one subscriber.

In Angular, the `HttpClient.get(URL)` method returns an observable. Initially, the observable is empty and no request is sent to the server at the URL. When the observable is subscribed, it sends a `get` request to the server. Even though, observables in general can return data gradually in parts, the `HttpClient.get()` method always returns the entire HTTP response in a

single step.

As a convention, a `$` is often appended to the names of observables. For example, `employees$`, `heroes$`.

An observable can be subscribed to in two ways - either by using the `async` pipe in the HTML template or by using a `Subscription` object in the view model code (`.ts` file).

```
<!-- HTML templqate -->
  <h2>{{selectedHero.name | uppercase}} Details</h2>
```

```
// .ts file
```

Angular Pipes

Angular borrows the concept of pipes from UNIX systems. In Angular HTML templates, content can be *piped* into an Angular pipe. The Angular pipe acts as a filter that can modify the content, or act on it in some other way. Examples of Angular pipes include `uppercase`, `lowercase` and `async`.

```
  <h2>{{selectedHero.name | uppercase}} Details</h2>
```

The `async` pipe is used with an observable. It subscribes to the observable and returns values from the observable.

```
<li *ngFor="let hero of heroes$ | async" >
```

The concept of pipes is also used with observables. The result of a subscription to an observable can be *piped* into various `RxJS` operators like `tap`, `map` and `catchError`.

```
    return this.http.get<Employee>(this.employeesDirectoryUrl +
        "/" + employeeNumber).pipe(
      tap(_ => this.messageService.log('fetched data:\n' +
              JSON.stringify(_).substr(0, 150) + '...')),
      //      tap(_ => this.log('fetched data:\n' +
        JSON.stringify(_, null, 4))),
      // No need to map, web service returns employee object only
      catchError(this.handleError<Employee>
        ('getEmployeesByEmployeeNumber', undefined))
```

```
    );
```

RxJS Operators

- **tap** is an `RxJS` operator that returns the values from the observable as they are, without any processing
- **map** `RxJS` operator can be used to process the value(s) returned by an observable and return a modified value
- **catchError** `RxJS` operator is used to handle errors that may arise in the asynchronous execution

Angular Routing

Angular routing is used to navigate between views. It maps paths to components to be displayed. At the time of creating the app using `ng new <app-name>`, it prompts us whether we would like to include support for routing in the app. If we respond with a `y` (yes), routing support is enabled in the app from beginning (It can be added to the app later also). It also creates a file `app-routing.module.ts` in the `app` subfolder.

Initially, `app-routing.module.ts` contains an empty list of routes.

```
const routes: Routes = [];
```

We can add routes to it as follows.

```
1  const routes: Routes = [
2    { path: '', redirectTo: '/index.html', pathMatch:
   'full' },
3    { path: 'employees', component: EmployeesComponent },
4    { path: 'employees/:id', component:
   EmployeeDetailComponent },
5    { path: 'employees/add/new', component:
   EmployeeAddComponent },
6    { path: 'employees/update/:id', component:
   EmployeeUpdateComponent },
7    { path: 'employees/delete/:id', component:
   DeleteEmployeeComponent },
8  ];
9
10 @NgModule({
11   imports: [RouterModule.forRoot(routes)],
12   exports: [RouterModule]
```

```
13 })
14 export class AppRoutingModule { }
```

We also need to write `imports: [RouterModule.forRoot(routes)],` and `exports: [RouterModule]` in the @NgModule decorator as shown on lines 11 and 12.

Paths are relative to the base URL of the application (e.g. `http://localhost:4200/`). `pathMatch: 'full'` ensures that the full path is matched against a rule, and not a part of it. Routes are matched in order and the first match is used. Hence, An empty path (only the base URL of the app) will match with the first route in the list and will be redirected to the file `/index.html`.

We can write `:variable_name` in the path. The value of that particular segment of the path can be retrieved using the `ActivatedRoute` service as follows:

```
import { ActivatedRoute } from '@angular/router';
    ...
  constructor(private route: ActivatedRoute) { }
    ...
  let id = this.route.snapshot.paramMap.get('id');
```

To display a component decided by routing, we write the following in our HTML template:

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

We can create a link to a particular route in the HTML template as follows:

```
<a routerLink="/employee/{{employee.employeeNumber}}">Details</a>
```

We can navigate to a particular route from code as follows:

```
import { Router } from '@angular/router';
    ...
  constructor(private router: Router) { }
    ...
  this.router.navigate(['employees']);
```

Note that the `navigate()` method needs an array of strings (routes).

RESTful Web Services

Classic Web Service: HTTP, XML, SOAP, WSDL, UDDI (deprecated)

REST

REpresentational State Transfer

RESTful Web Services: HTTP

All entities are represented as resources. Each resource is associated with a unique URI / URL.

E.g.

List of employees (directory / list URL): http://localhost:8080/employee

A particular employee (employeeNumber 9999): http://localhost:8080/employee/9999

CRUD Operations

CRUD	Description	SQL
Create	Create a new row	INSERT
Retrieve	Fetch row(s)	SELECT
Update	Modify a row	UPDATE
Delete	Remove a row	DELETE

HTTP Verbs / Methods

Function	Verb / Method	Use in REST
Retrieving a list of resources	GET	directory URL
Retrieving a particular resource	GET	resource URL
Inserting a new resource	POST	directory URL
Updating a resource	PUT	resource URL
Deleting a resource	DELETE	resource URL

Stateless protocol (There is no concept of session)

- Authentication

- Web tokens (e.g. JWT - JSON Web Tokens)
- OAUTH
- OAUTH2

REST clients

- Postman

REST client extensions for browsers

FireFox : RESTClient, RESTED

Chrome / Edge : Boomerang, RESTED

CORS Policy

SOP
: Same Origin Policy. A web browser security policy under which a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin.

CORS
: Cross Origin Resource Sharing.

CDN
: Content Distribution Network.

Same-origin policy

The same-origin policy (SOP) is an important concept in the web application security model. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of URI scheme, host name, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.

This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions. A strict separation between content provided by unrelated sites must be maintained on the client-side to prevent the

loss of data confidentiality or integrity.

It is very important to remember that the same-origin policy applies only to scripts. This means that resources such as images, CSS, and dynamically-loaded scripts can be accessed across origins via the corresponding HTML tags (with fonts being a notable exception). Attacks take advantage of the fact that the same origin policy does not apply to HTML tags.

JavaScript Notes

JavaScript Truthiness

If the value is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), it is treated as false. All other values, including any object, an empty array ([]), or the string "false", are treated as true.

Equality operator (==) v/s strict equality (identity) operator (===) in JavaScript

Strict equality operator always considers values of different types to be different. Equality operator tries to coarse (convert) values of different types into a single type and then compares the results.

Equality operator only looks at the values of the operands. Strict equality operator first compares the data types of the operands. If the data types are different, the operands are considered as "not equal". If the data types of both the operands are same, then only the strict equality operator looks at their values.

In TypeScript, values of different types cannot be compared (but there are union types and the any type)

let no1: any = 0; let str1: any = "";

// Equality operator let result1 = (no1 == str1); console.log(result1);

// Strict equality (identity) operator let result2 = (no1 === str1); console.log(result2);

JavaScript Template Strings

Javascript template strings are enclosed in backquotes (`  `). They may

contain JavaScript variables and expressions in `${ }`. For example, if value of variable `user` is `"Abc"`, then `` `Welcome back ${user}!` `` evaulates to `"Welcome back Abc!"`.

Lambda Function (functional programming syntax)

```
// JavaScript
let x = 10;
let y = 20;
let z;

// Without using Lambda functions
function add(a, b) {
    return a + b;
}

z = add(x, y);
console.log(z);

// Defining a JavaScript function as a variable of type function
let addFunction = function(a, b) {
    return a + b;
}

z = addFunction(x, y);
console.log(z);


// Defining a  Lambda function
let addLambda = (a, b) => { return a + b; }

z = addLambda(x, y);
console.log(z);

// Using an anonymous Lambda function
// z = ( (a, b) => { return a + b; } )(x, y);
z = ( (a, b) => a + b )(x, y);
console.log(z);

/////// OUTPUT ///////
30
30
30
30
```

```
1 // JavaScript
2 // Using Regular Functions
3 let x = 10;
```

```
let y = 25;
let z;

// operation is a parameter of type function
function calculate(a, b, operation) {
    return operation(a, b);
}

function add(a, b) {
    return a + b;
}

function sub(a, b) {
    return a - b;
}

function mul(a, b) {
    return a * b;
}

function div(a, b) {
    return a / b;
}

z = calculate(x, y, add);
console.log(z);

z = calculate(x, y, sub);
console.log(z);

z = calculate(x, y, mul);
console.log(z);

z = calculate(x, y, div);
console.log(z);

// JavaScript
// Using Lambda Functions

// operation is a parameter of type function
// let calculateLambda = function(a, b, operation) {
//     return operation(a, b);
// }
calculateLambda = ( (a, b, operation) => operation(a, b)
);

z = calculateLambda(x, y, (a, b) => a + b);
console.log(z);

z = calculateLambda(x, y, (a, b) => a - b);
console.log(z);

```

```
z = calculateLambda(x, y, (a, b) => a * b);
console.log(z);

z = calculateLambda(x, y, (a, b) => a / b);
console.log(z);

/////// OUTPUT ///////
35
-15
250
0.4
35
-15
250
0.4
```

TypeScript Introduction and Reference

Inside a method body, it is mandatory to access fields and other methods of the same class via `this.`. An unqualified name in a method body will always refer to something in the enclosing scope:

```
let x: number = 0;

class C {
  x: string = "hello";

  m() {

    // The following is OK
    this.x = "world";

    // This is trying to modify 'x' from line 1, not the class property
    x = "world";
    // ERROR: Type 'string' is not assignable to type 'number'.

  }
}
```

Decorators

A *decorator* (known as *annotation* in some other programming languages) is used to provide meta data or additional data about a programming language construct like a class, a data member, a property, a method, etc. A decorator

can have optional / compulsory attributes / parameters also. In TypeScript, the decorator is written immediately before a class, property, method, etc. and it provides additional information about programming construct that immediately follows it. The syntax for decorator is `@Decorator({attributes})`

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'classicmodels-M3-B2';

  constructor() {
    console.log('Root component constructor()...');
  }
}
```

Union types in TypeScript

A union type is a type formed from two or more other types, representing values that may be any one of those types.

```
id: number | string;
reportsTo: number | null;
```

Structural Subtyping and Type Compatibility in TypeScript

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members. This is in contrast with nominal typing. Consider the following code:

```
interface Pet {
  public void play();
}

// Does not explicilty implement the Pet interface
class Dog {
  public String name;

  Dog(String name) {
      this.name = name;
  }

  public void play() {
```

```
14      System.out.println(this.name + " playing with
    ball");
15    }
16  }
17
18  public class Test
19  {
20      public static void main (String[] args)
21      {
22          Pet pet = new Dog("Jawa");
23          pet.play();
24      }
25  }
```

```
/////// COMPILER OUTPUT ///////
javac Test.java
Test.java:22: error: incompatible types: Dog cannot be converted
to Pet
        Pet pet = new Dog("Jawa");
                  ^
1 error
```

```
interface Pet {
  name: string;
  play();
}

// Does not explicilty implement the Pet interface
class Dog {
  name: string;
  play() {
    console.log(this.name + ' playing with ball');
  }
}

// Not allowed in languages like Java and C#
// Allowed in TypeScript because of structural typing
let pet: Pet = new Dog();
pet.name = 'Jawa';
pet.play();
/////// OUTPUT ///////
Jawa playing with ball
```

Additional Notes

To increase the font size of all windows (not just editor) in VS Code:

File -> Preferences -> Settings -> (search for "Zoom") -> Window -> Zoom Level: Set to a value like 1.2

To update node to latest version:

sudo npm install -g n

sudo n stable

-- OR --

sudo n latest

if n is not found, which n shows the path where it is installed (/home/jignesh /.npm-global/bin/n).

A specific version can bne instyalled using:

sudo n 8.9.0

To keep npm updated:

sudo npm i -g npm
