WFS Notes

1. Java 7 and Java 8
2. Spring
3. Spring Microservices
4. Angular Framework

Java has done major changes in Java 5, Java 7 and Java 8

Java 5 features

1. for each loop
2. Enums
3. Generics
4. Var-Args
5. Static Imports
6. Annotations

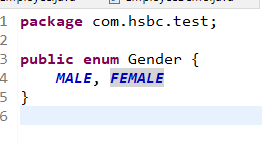
Enums: Fixed set of constants, so that at compile time itself you can verify the valid values, ex:

Gender must be male or female

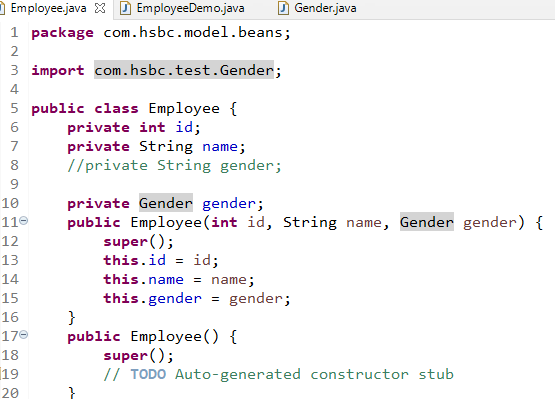
Account must be Savings, Current, Recurring, Fixed Deposit

Departments must be Accounts, Sales, Finance,..

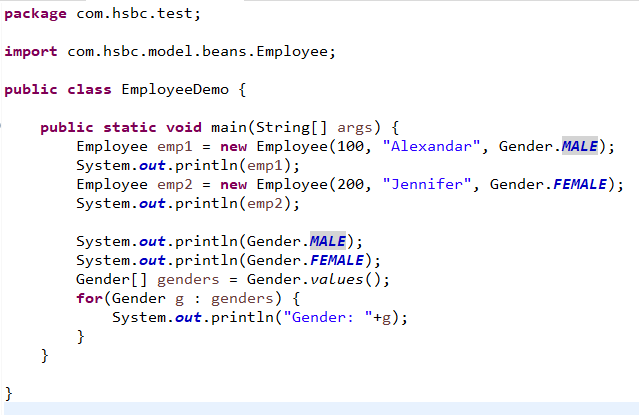
Gender.java



Employee.java



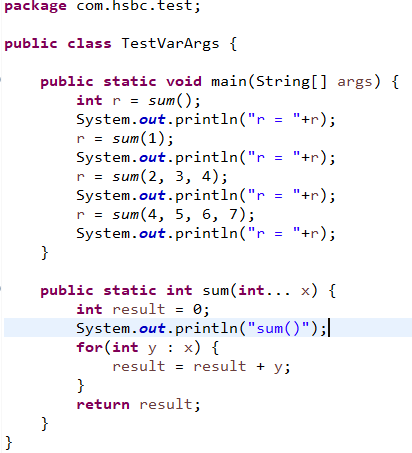
EmployeeDemo.java



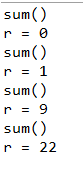
Varying Arguments

It can accept 0 or more arguments

int sum(int... x)



Output:



Java 7 Features

1. Diamond Operator
2. Multi-Catch block
3. Resource Management with try-with resource syntax
4. Strings in Switch case
5. Underscores in numbers

Diamond Operator <>:

Before Java 7:

List<Integer> list = new ArrayList<Integer>();

From Java 7:

List<Integer> list = new ArrayList<>();

Multi-Catch Statements:

In one catch you can have one or more exceptions to handle

catch(ArithmeticException | NumberFormatException e) { }

Resource Management with try-with resource statement

This features automatically closes the resources like files, buffers, databases etc.

It automatically flushes the streams while performing write operations

Syntax:

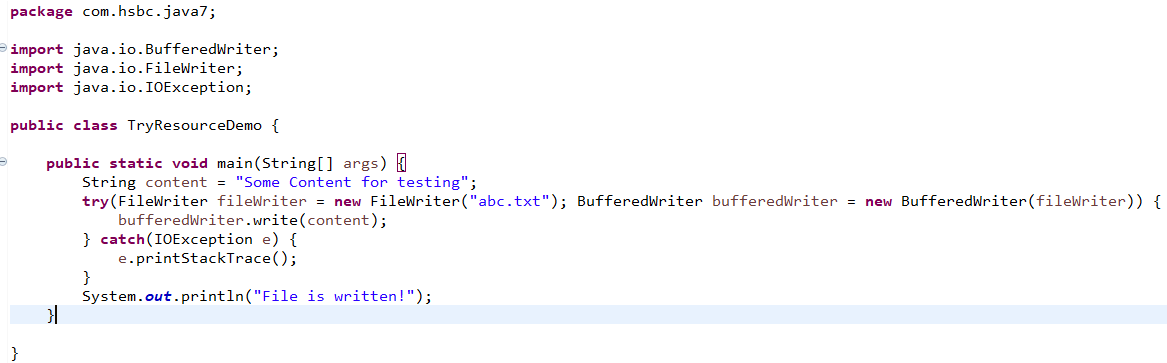
try (resource instances; resource instances) {

}

Example:

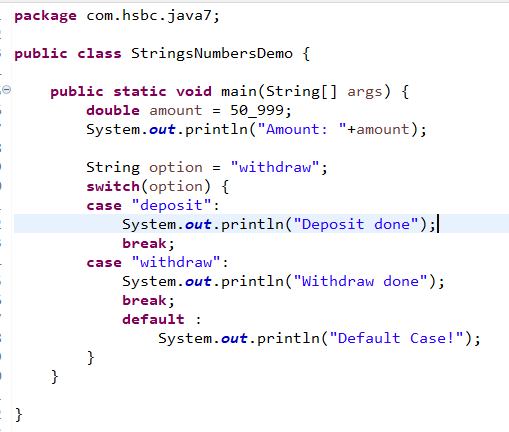
try (fw = new FileWriter(“abc.txt”)) { }

This makes the Java to automatically perform close after performing the operations inside the try



Above code automatically closes the streams

Strings in Switch & Underscores in Numbers



Java 8 Features:

1. New Date & Time API’s
2. Static & Default methods inside interface
3. Functional Interface
4. Lambda Expressions
5. Stream API’s

New Date & Time API’s

Java introduced 3 main classes to work on Date, Time and DateTime

1. LocalDate
2. LocalTime
3. LocalDateTime

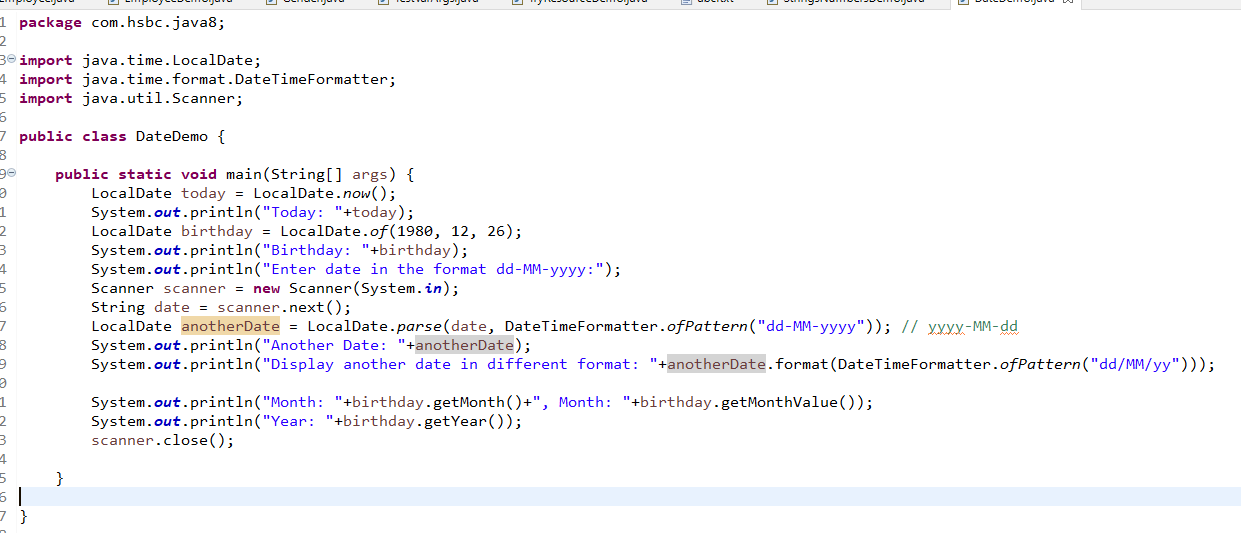
All these 3 classes have some static methods to create date & time instances

now(): to create current date or time instance

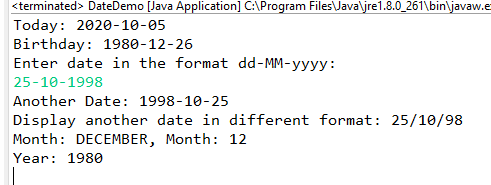
of(): to create specific date or time instance

parse(): to take a string and convert to date & time instance

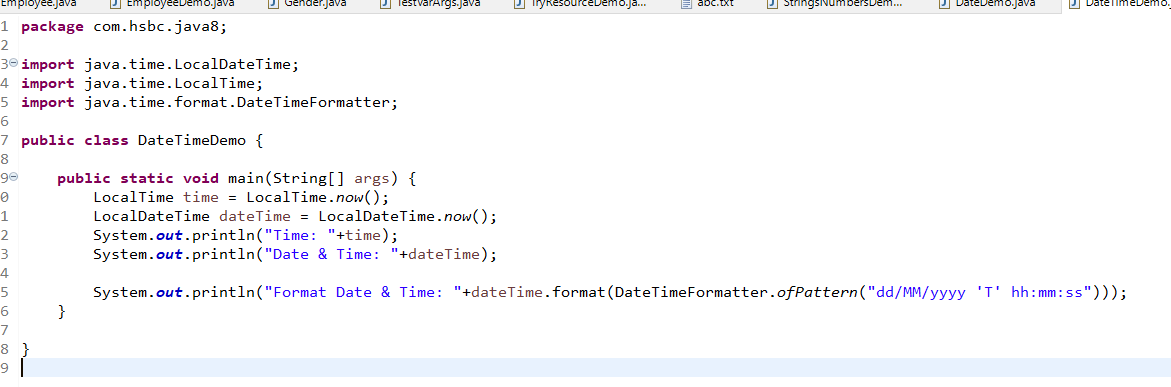
LocalDate Demo



Output:



LocalTime & LocalDateTime also works the same way



Output:

Time: 11:45:17.188

Date & Time: 2020-10-05T11:45:17.189

Format Date & Time: 05/10/2020 T 11:45:17

Changes in the interface

You can have methods with body in the interface

1. default methods: it will have some default implementations but not necessary to override
2. static methods: you can call directly from the interface name

Lambda Expression:

It is a simplified form of implementing anonymous class, it has to be applied only on the interfaces having single abstract method (Functional Interface)



Lambda expressions are better when the statement is single line expression, but it also allows you to have multiline statements

() -> 10 // a method that returns an int

() -> “Hello” // a method that returns a String

(x, y) -> x + y // a method that returns addition of x & y

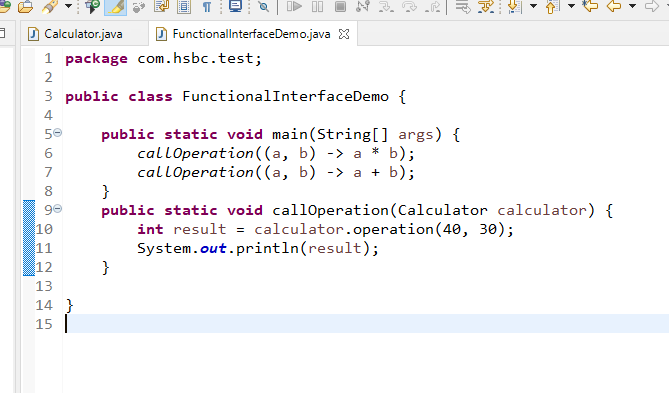
() -> { System.out.println(“...”); return 10; }

() -> System.out.println() // a method returns void

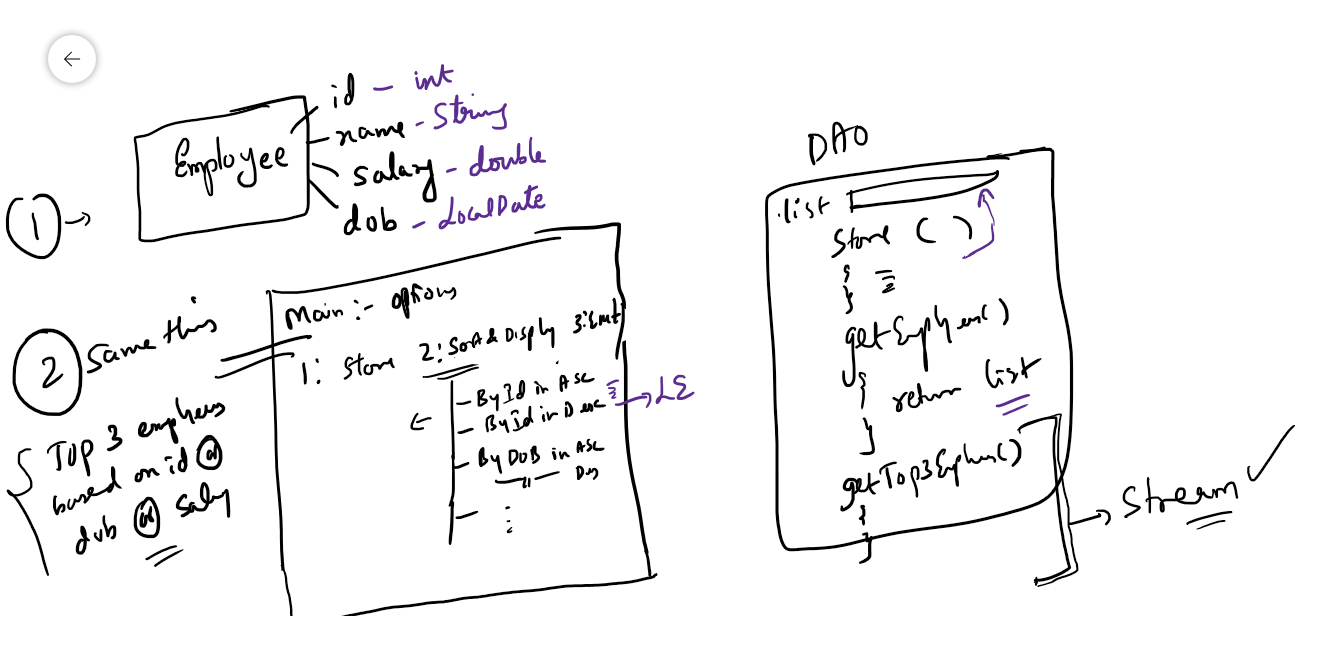
Note: return statement is required when a method has a return type if you write more than one line in lambda expression

Functional Interface:

It allows you to pass code directly to a method, it is an interface with only one abstract methods.



Exercise:



1st requirement: get all the employees added into the collection

2nd requirement: get only the top 3 employees sorted based on id, salary, dob

Java Streams:

Streams are collection of data that you want to operate while processing the collection, Java Streams processes the collection of data in a declarative way like SQL.

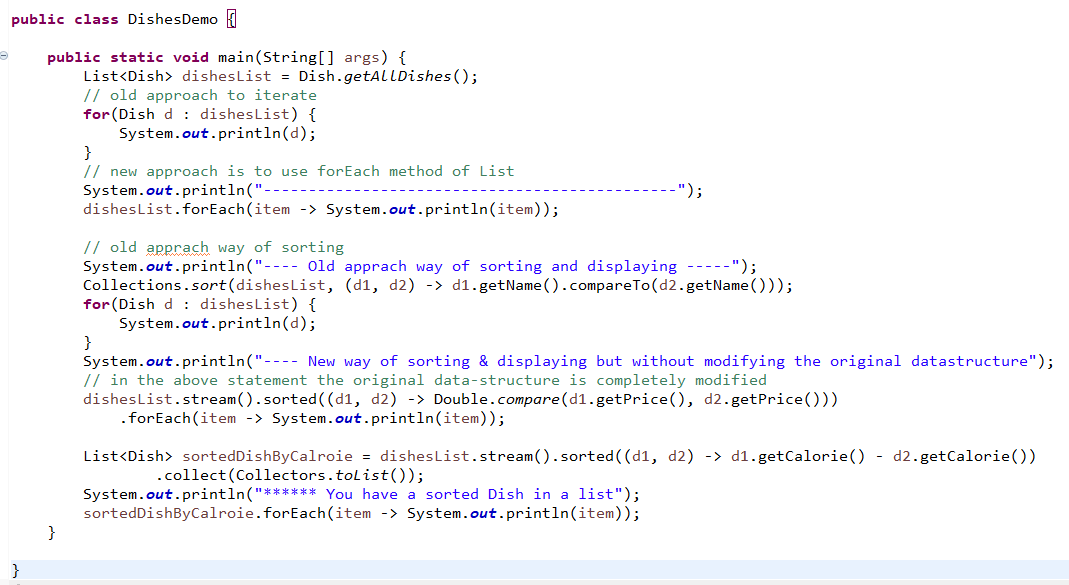
SQL statements are declarative statements they are easier to write, to select items, to filter items, to sum items

select \* from employee;

select \* from employee order by name;

select \* from employee order by name desc;

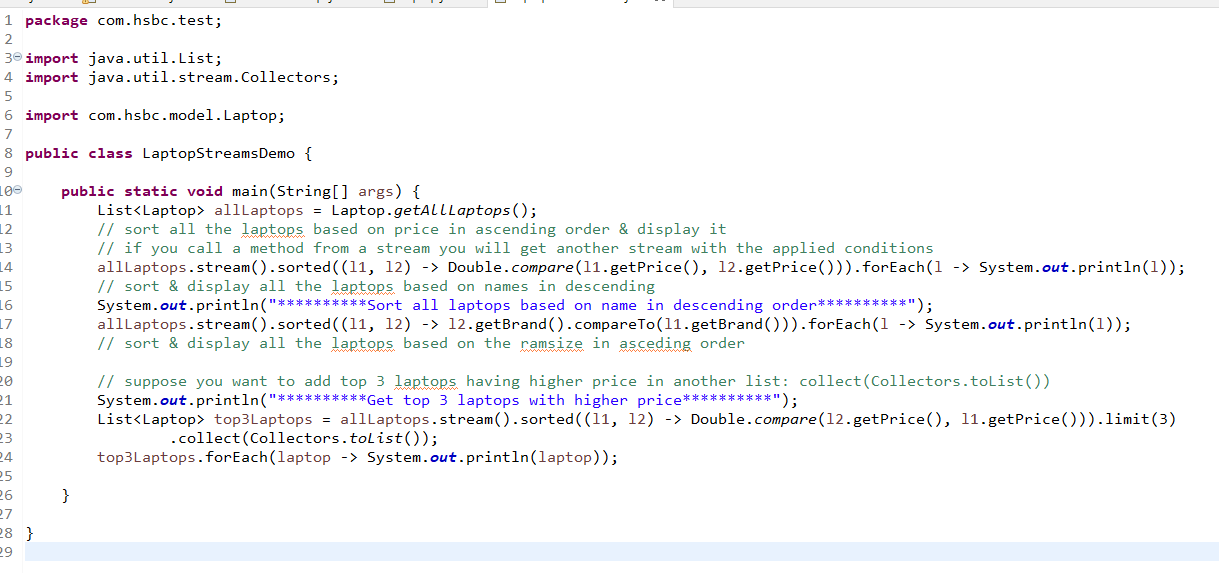
You can simplify sorting, filtering, iterating in the streams



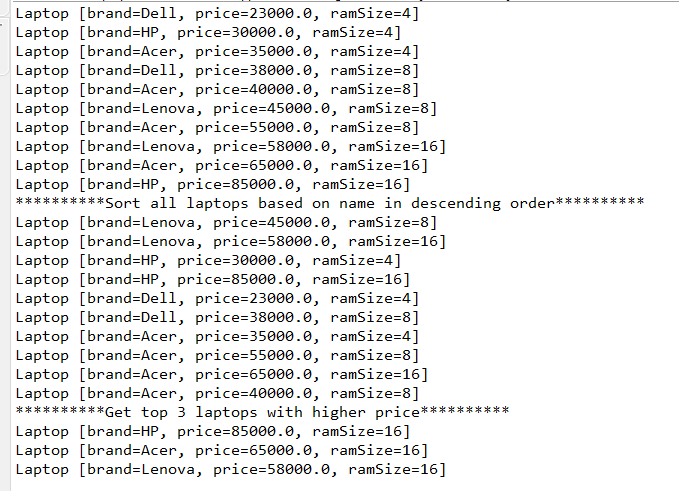
Streams has many methods that can be chained that gives another stream and some methods also used at the end which is called as terminal operation.

There are two type of operations you can do on streams

1. Intermediate
   1. sorted
   2. filter
   3. distinct
   4. limit
   5. map
2. Terminal
   1. forEach
   2. count
   3. collect

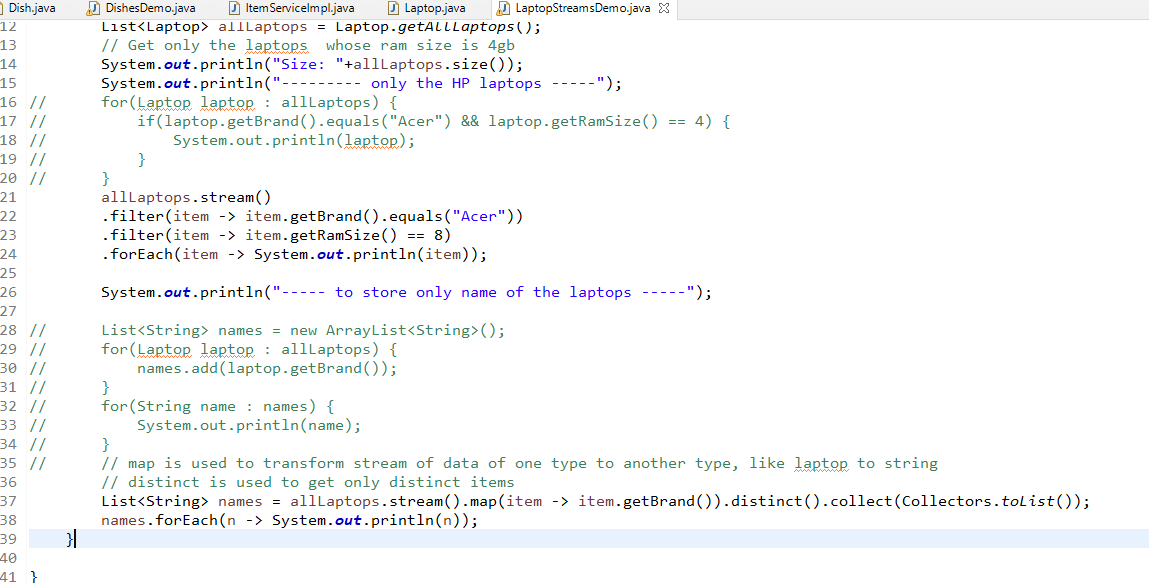


Output:



filter: it is used to filter the data from a stream by applying some condition.

* get laptops of specific brand name
* get laptops of specific ram size



Spring Framework

Framework makes developers to develop complex applications in a simpler way, it will take care of lot repeating tasks like Exception Handling, Type Conversion, Design Patterns, Object creations, configurations and so on

Spring Framework is one of the java framework which is very popular because you develop many kinds of applications like standalone, web, mobile, microservices and so on.

Spring Framework or any other frameworks use lot of libraries to make the task simple hence you must download those libraries from the internet, which is why it’s recommended to use maven project which has a feature of downloading the libraries from the internet.

Spring provides many modules for different technologies

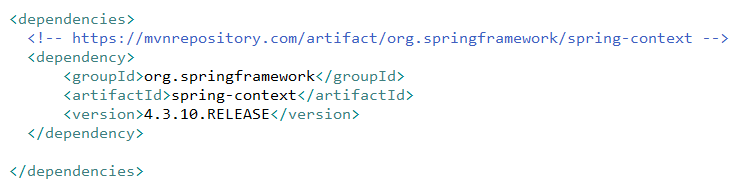
1. Spring Core : Fundamentals
2. Spring Web : Web applications
3. Spring Data JPA : Database purpose
4. Spring REST : Application Integration
5. Spring Boot : Simplifies spring application development

Spring Core: The basic unit of spring core is dependency injection.

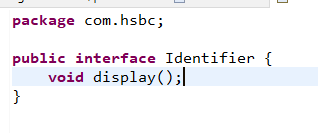
Dependency Injection: It supplies dependency of an object to another object, so that you don’t have to create object or initialize the object in another code.

Spring Framework maintains all the objects in its container called Spring Context, it initializes the objects by looking at the xml configuration.

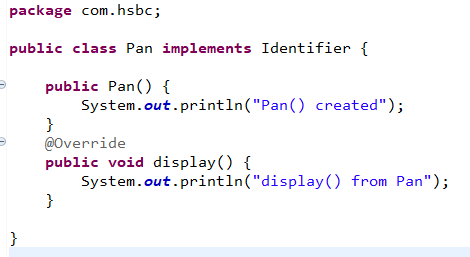
pom.xml



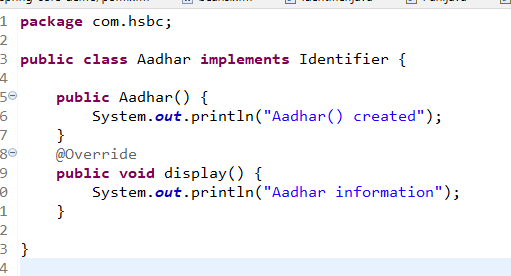
Identifier.java



Pan.java



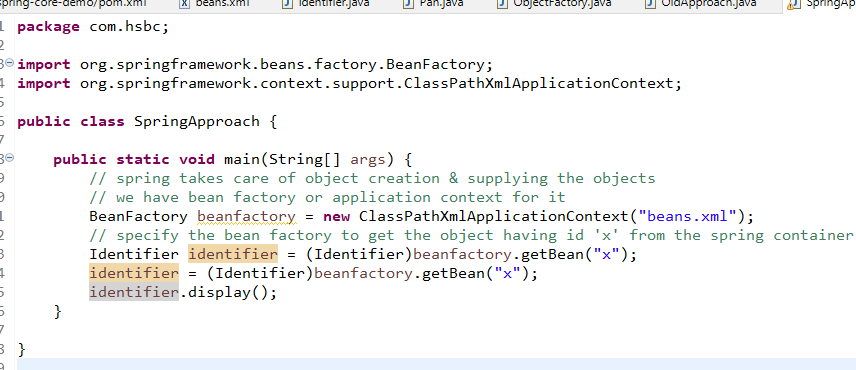
Aadhar.java



OldApproach.java



SpringApproach.java



Spring container can be accessed through

1. BeanFactory
2. ApplicationContext

BeanFactory is the super type for ApplicationContext.

You can also initialize the object variables in the xml file.

Spring Container maintains all the objects and it supplies the dependencies to other objects.

Spring Core or IoC: Inversion of Control, it is a mechanism where object creation is inverted, like instead of code creating the object, framework creates the object i.e, container will maintain the object.

There are three ways you can achieve dependency injection

1. setter injection
2. constructor injection
3. @Autowired

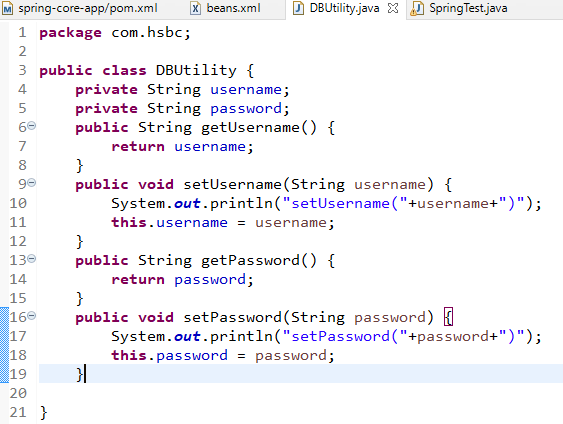
setter injection is a way where objects are supplied by calling setter methods, i.e., variables of an object is initialized via setter this is done using <property> tag in the xml

constructor injection is a way where objects are supplied by calling constructor, i.e., variables of an objects is initialized via constructor

@Autowired is a way where objects are directly supplied to the variable without using constructor or setter

Setter Injection Demo

DBUtility.java



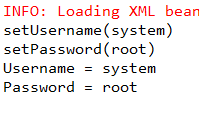
beans.xml



SpringTest.java



Output:



Constructor Injection Demo

DBUtility.java



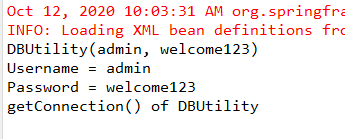
beans.xml



SpringTest.java



Output:



Note: getConnection() should be treated as a method that establishes the connection on a particular database & its called usually in DAO layer

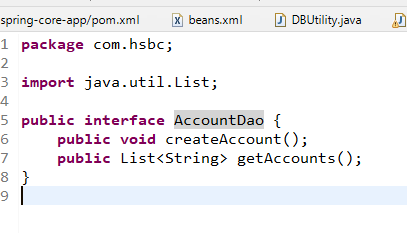
The above examples supply values to the object, but you can also supply dependent object to the object.

<property name = “variable” ref = “idName”>

<constructor-arg index = “...” ref = “idName”>

Supplying an object to another object

AccountDao.java



JdbcBackedDao.java



beans.xml

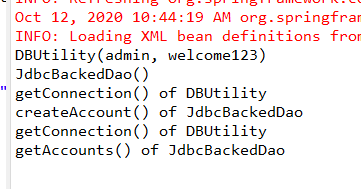


Note: Client program doesn’t need DbUtility instead it can take DAO instance & call the methods

SpringTest.java



Output:



Assignment:

1. Pass DBUtility through constructor injection
2. Implement another DAO like HibernateBackedDAO for the AccountDAO perform the required dependency injection the same way and call the createAccount() & getAccounts() method
3. Perform a dependency injection by adding another layer i.e., Service Layer (AccountService) and make framework to supply DAO instance to the service layer and from client code(main method) you call the methods of service layer

Hint: AccountService interface with

* 1. createAccount
  2. getAccounts
  3. getAccountsSortByName

Spring Annotation based configuration:

It simplifies configuration of spring features so that xml configurations can be reduced

XML based way  
<bean id = “” class = “”>

Annotation based way:

@Component  
public class A { }

@Component specifies spring to instantiate the class

There are other annotations you can use

@Repository, @Service, @Controller, @RestController and so on

All these annotations are part of @Component, but you can use them based on the layer type

@Repository can be used in the DAO

@Service can be used in the Service

@Controller can be used in the Controller

Supplying dependencies can also be done through annotations using @Autowired on top the variable

XML based way:

<bean ....>   
 <property name = “..” ref = “..”>

Annotation based way:

@Autowired

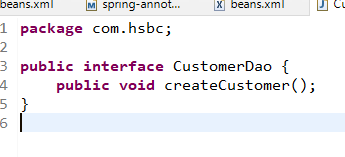
private AccountDao dao;

To configure spring to support annotations you must use one tag in the xml called <context:component-scan>

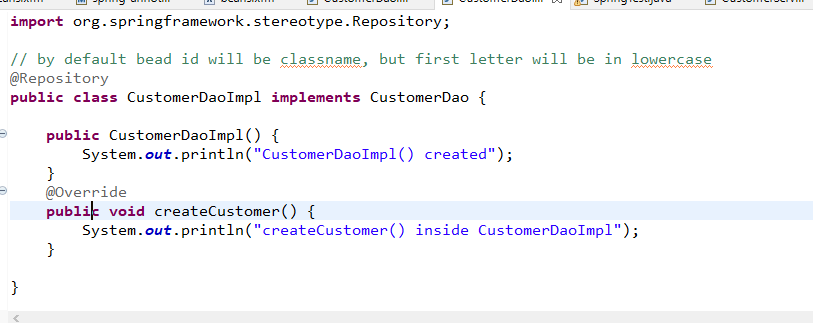
beans.xml



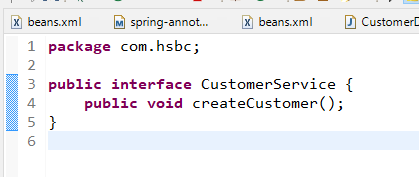
CustomerDao.java



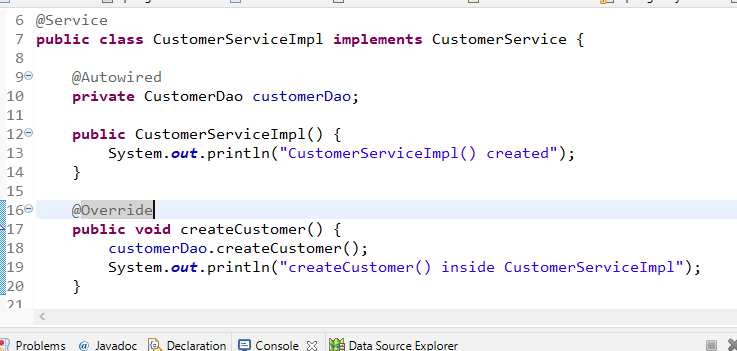
CustomerDaoImpl.java



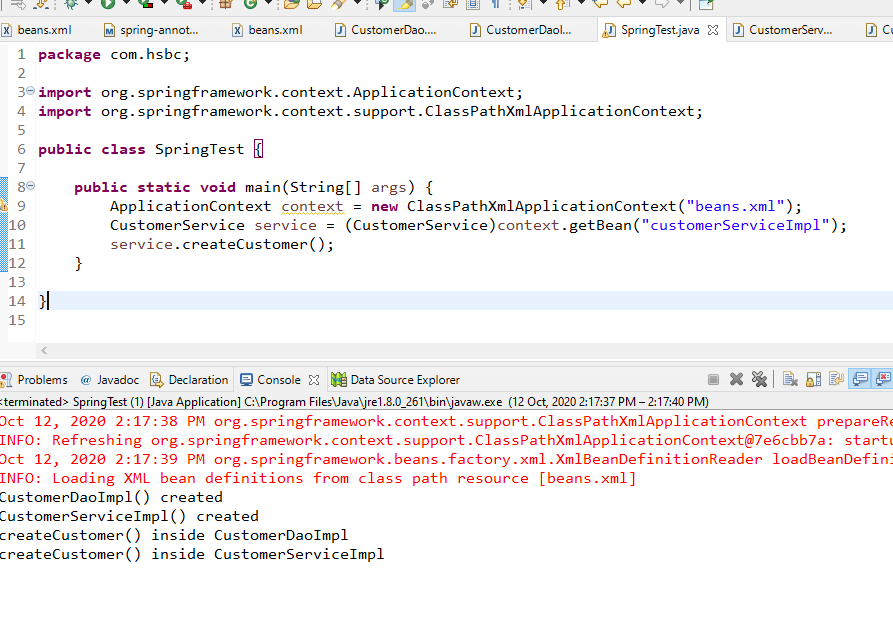
CustomerService.java



CustomerServiceImpl.java



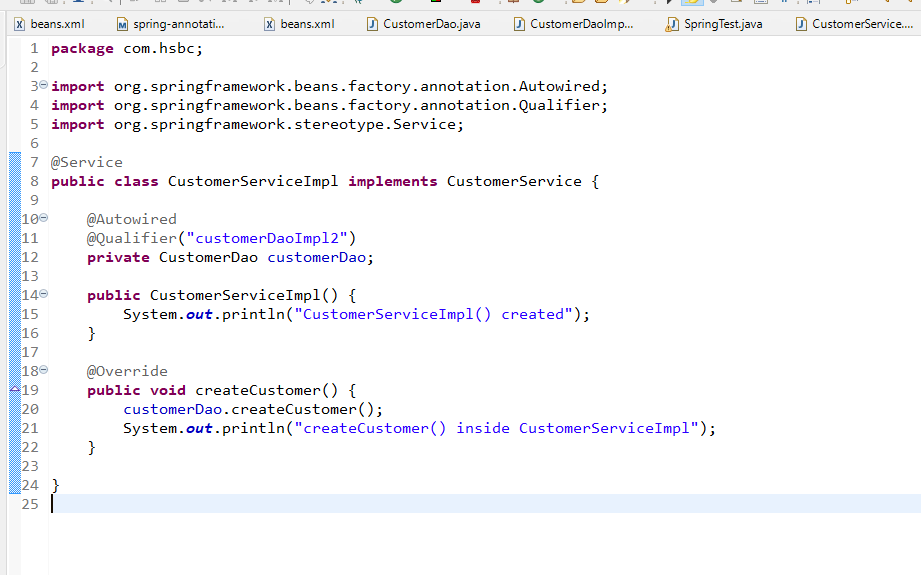
SpringTest.java



@Autowired: it automatically injects the object based on the type of the variable,

*CustomerDao customerDao* variable has *@Autowired*, spring framework checks for the object that implements CustomerDao, but the condition is there must be only one object of that particular type, if found more than one, spring raises error UnsatisfiedDependencyException, to resolve you can use *@Qualifier* or you can configure spring to have only one object of that type

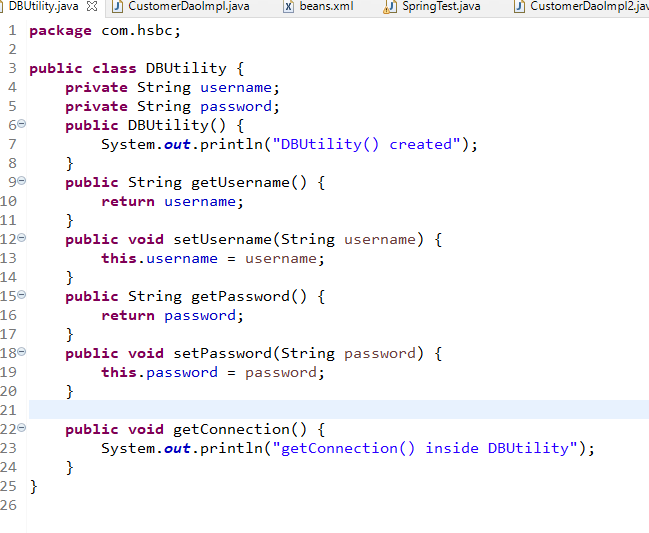
CustomerServiceImpl.java



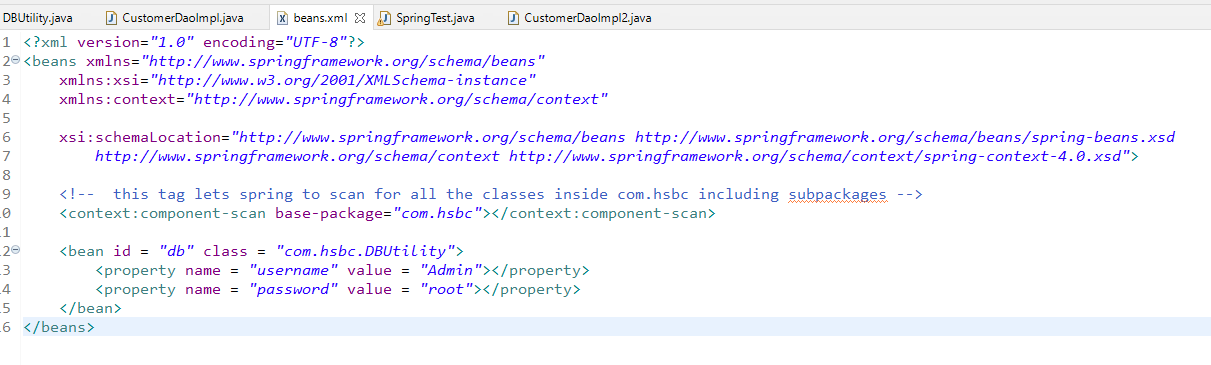
Since there are 2 Dao implementations we must use @Qualifier

Can we configure the class in the xml and inject the object using annotation

DBUtility.java

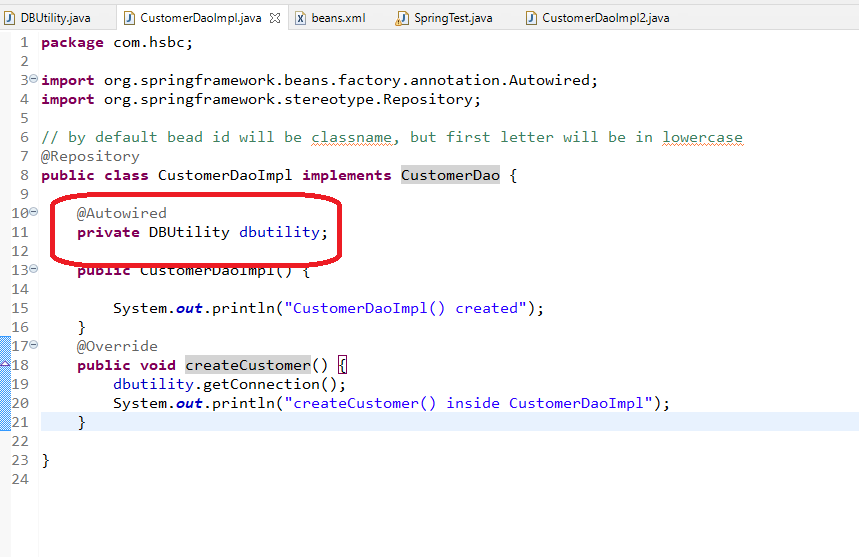


beans.xml



Note: When DBUtility has some properties you can initialize those properties in XML or suppose you don’t have DBUtility class with any stereo type annotations and it is provided by some third party vendors, then you can configure in the xml only

CustomerDaoImpl



Spring MVC Module

It allows you to develop web applications in a simplified way using spring features.

Spring MVC uses FrontController it can take all the requests coming to the web application & route to the appropriate controller, the front controller name is DispatcherServlet.

DispatcherServlet task:

1. it maps the request to the appropriate controller
2. it takes care of creating ApplicationContext
3. it takes care of mapping the response to appropriate view (JSP)

You must add spring mvc library to your project, it automatically adds dependent library i.e., spring-context

web.xml



DispatcherServlet is a front controller, which creates ApplicationContext object and specifies the xml file name should be *<servlet-name>-servlet.xml*

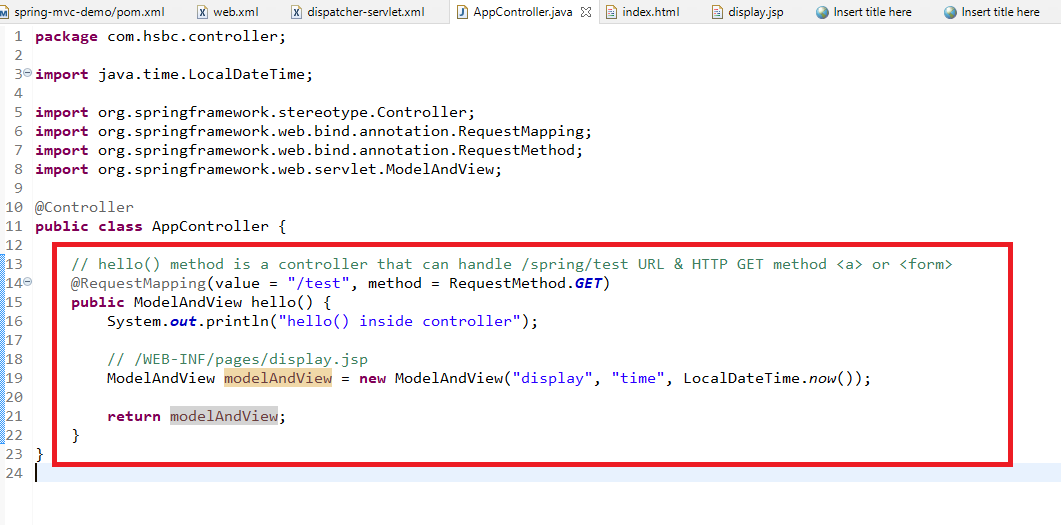
i.e., dispatcher-servlet.xml

dispatcher-servlet.xml



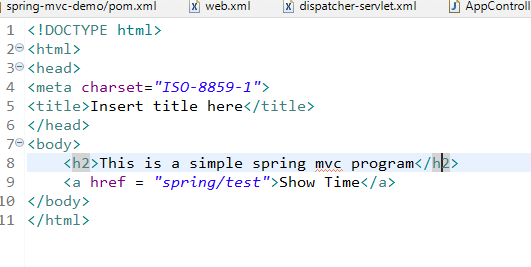
InternalResrouceViewResolver locates the view by adding prefix & suffix to the view name coming from the ModelAndView instance.

AppController.java



*@Controller* configures the class to be controller & its instantiated at the time component scan mentioned in the dispatcher-servlet.xml

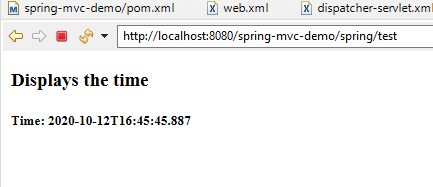
WebContent/index.html



WebContent/WEB-INF/pages/display.jsp



Output:



Implementing simple MVC program with Spring MVC

UserDao.java

**package** com.hsbc.model.dao;

**public** **interface** UserDao {

**public** String fetchUserById(**int** id);

}

UserDaoImpl.java

**package** com.hsbc.model.dao;

**import** org.springframework.stereotype.Repository;

@Repository

**public** **class** UserDaoImpl **implements** UserDao {

@Override

**public** String fetchUserById(**int** id) {

**if**(id == 100)

**return** "MS Dhoni";

**return** "Virat Kholi";

}

}

UserService.java

**package** com.hsbc.model.service;

**public** **interface** UserService {

**public** String fetchUser(**int** id);

}

UserServiceImpl.java

**package** com.hsbc.model.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.hsbc.model.dao.UserDao;

@Service

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

**private** UserDao userDao;

@Override

**public** String fetchUser(**int** id) {

**return** userDao.fetchUserById(id);

}

}

AppController.java

**package** com.hsbc.controller;

**import** java.time.LocalDateTime;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestMethod;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.servlet.ModelAndView;

**import** com.hsbc.model.service.UserService;

@Controller

**public** **class** AppController {

@Autowired

**private** UserService service;

// hello() method is a controller that can handle /spring/test URL & HTTP GET method <a> or <form>

@RequestMapping(value = "/test", method = RequestMethod.***GET***)

**public** ModelAndView hello() {

System.***out***.println("hello() inside controller");

// /WEB-INF/pages/display.jsp

ModelAndView modelAndView = **new** ModelAndView("display", "time", LocalDateTime.*now*());

**return** modelAndView;

}

@RequestMapping(value = "/user", method = RequestMethod.***POST***)

**public** ModelAndView getUser(@RequestParam("userid") **int** id) {

String username = service.fetchUser(id);

ModelAndView modelAndView = **new** ModelAndView("demo", "name", username);

**return** modelAndView;

}

}

demo.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

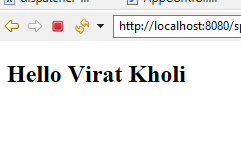
<body>

<h2>Hello ${name}</h2>

</body>

</html>

Output:



Controller is a method that can handle the request mentioned in the @RequestMapping, controller must return either

1. ModelAndView: view name, key and model
2. String: view name

Showing Java object in JSP

UserDaoImpl.java

**package** com.hsbc.model.dao;

**import** org.springframework.stereotype.Repository;

**import** com.hsbc.model.beans.User;

@Repository

**public** **class** UserDaoImpl **implements** UserDao {

@Override

**public** String fetchUserById(**int** id) {

**if**(id == 100)

**return** "MS Dhoni";

**return** "Virat Kholi";

}

@Override

**public** User fetchUserByName(String name) {

User user = **new** User();

user.setUsername(name);

user.setAge(33);

**return** user;

}

}

UserServiceImpl.java

**package** com.hsbc.model.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.hsbc.model.beans.User;

**import** com.hsbc.model.dao.UserDao;

@Service

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

**private** UserDao userDao;

@Override

**public** String fetchUser(**int** id) {

**return** userDao.fetchUserById(id);

}

@Override

**public** User fetchUser(String name) {

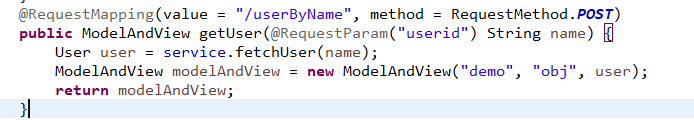
// **TODO** Auto-generated method stub

**return** userDao.fetchUserByName(name);

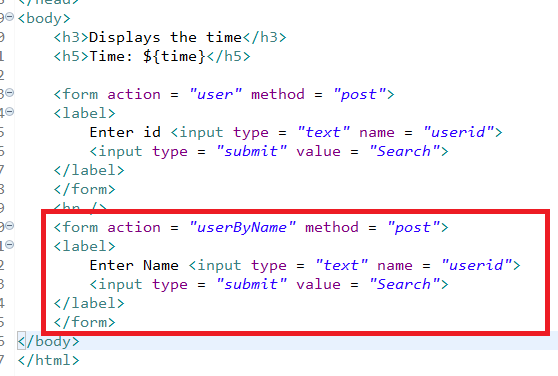
}

}

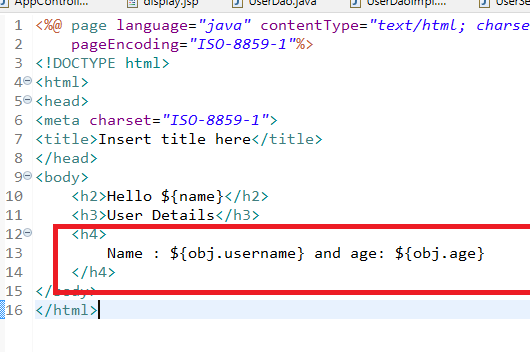
AppController.java



display.jsp



demo.jsp



Spring REST:

It is used develop Online API’s, which can be called from different languages.

REST stands for REpresentational State Transfer.

Representational: Data representation like JSON/XML

State: Data

Transfer: sending & receiving the data

HTML is understood by browser

JSON/XML is understood by programs, programs will decide where to show those data either on Browser or Mobile screen or ATM monitor or Swiping card devices

To create a RESTful Webservice you must use @RestController

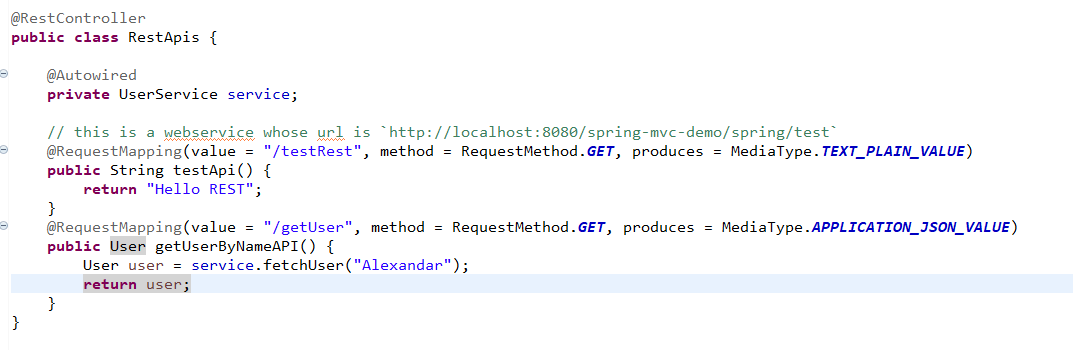
You can convert Java objects to JSON and vice versa

Note: for json type you must use one separate jar called Jackson databinding

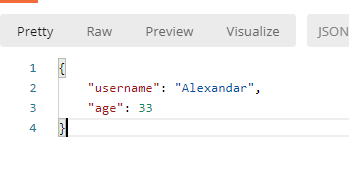
pom.xml



RestApi.java



Output:



Note: Alexandar is hardcoded, but you can pass the value at runtime using path variable

/getUser/{name}

Now path can be /getUser/Alexandar

/getUser/Bruce

/getUser/Charles

Multiple path variables:

employee/{id}/department/{dept}

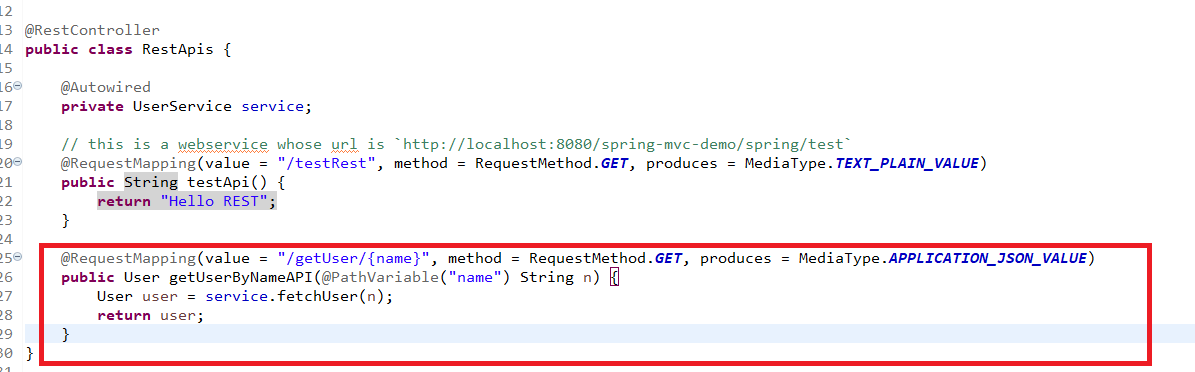
Now path can be

employee/100/department/10

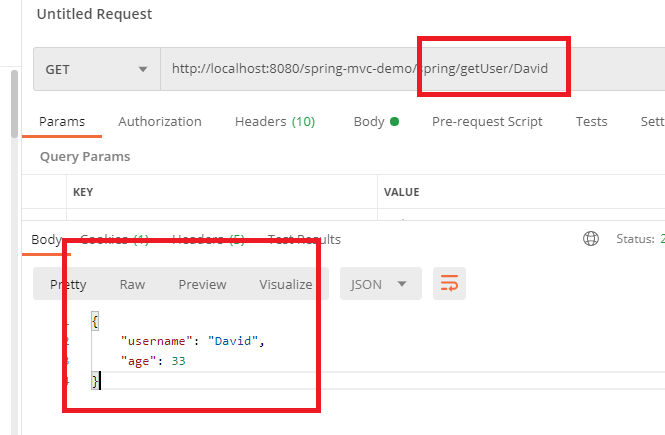
employee/200/department/20

@PathVariable(“id”)  
@PathVariable(“dept”)

RestApi.java



Output:



Spring Boot:

It simplifies spring configurations for developers, you can easily develop spring applications with the help spring boot.

Spring boot will take care providing all the basic setup for the projects.

@SpringBootApplication: it enables spring to auto-configure everthing when you add starter libraries of spring boot, we have many starter libraries in spring boot

spring-web starter: web application

spring-data-jpa starter: database interaction

spring-test starter: testing purpose

Spring boot uses one configuration file called *application.properties*, you can mention datasource, server configurations and many other configurations

Suppose you want to change port number

server.port = 9090

Suppose you want to mention datasource informations

datasource.username = db-username

datasource-password = db-password

For more informations on properties you can visit spring website

<https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/common-application-properties.html>

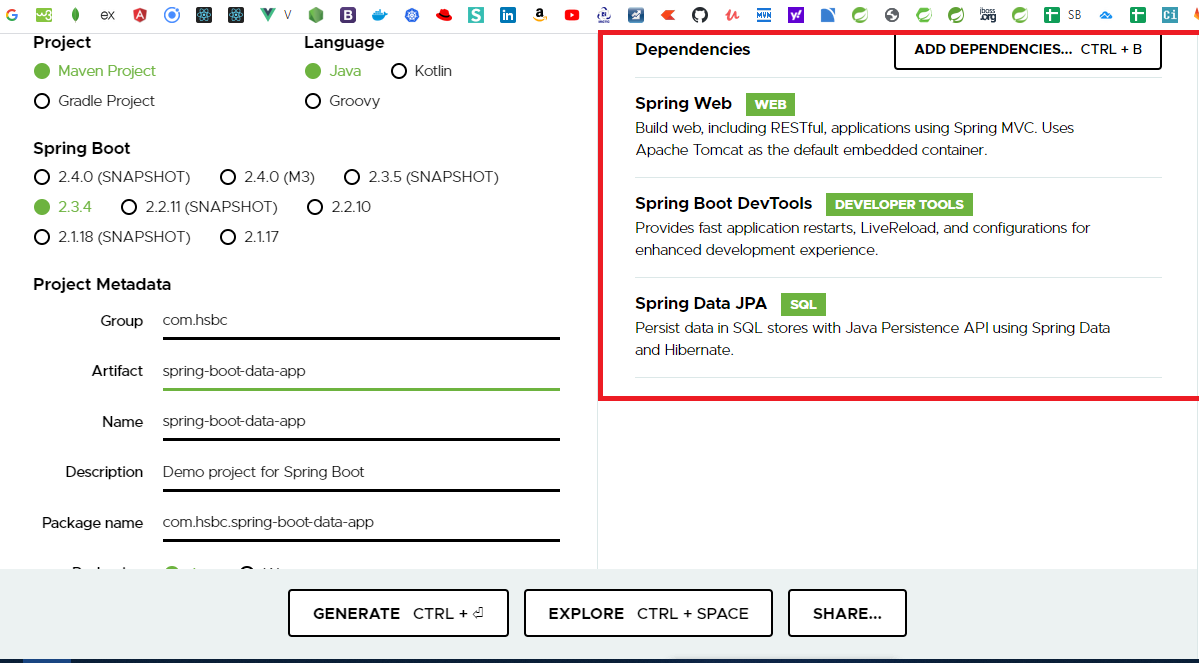
Spring Data Jpa starter: it is used to provide configurations for datasource

This allows you to directly map java objects to the table, so that you don’t have to write any queries to perform any CRUD operations, you can perform database operations in one line.

For this to happen you must create entity class which maps java class to the table & their properties to the columns of the table.

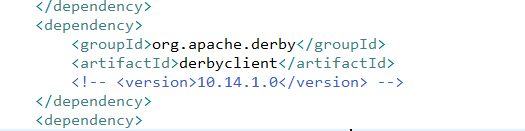
@Entity  
@Table(name = “user\_table”)  
public class User {   
 @Column(“user\_id”)  
 @Id  
 private int userId;  
   
 @Column(“username”)  
 private String name;  
  
 ..  
}

save(u); // u is a user object that will be mapped to user\_table



Note: jdbc driver jar wouldn’t be available you must add manually from maven repository

Add derby client

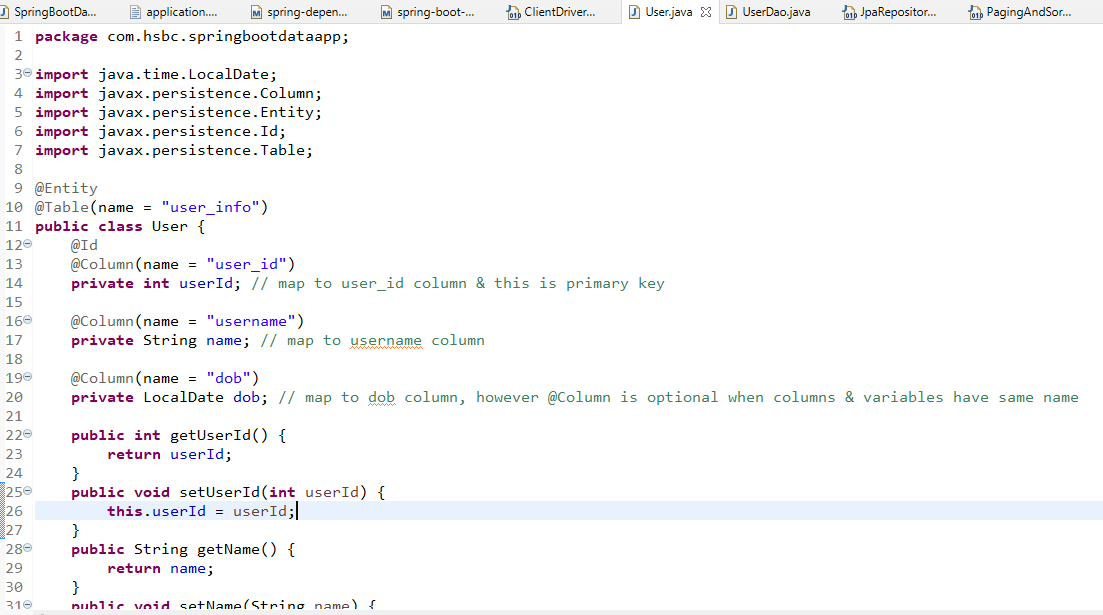


@SpringBootApplication: will auto-configure the datasource, it will immediately try to connect to the database using spring data jpa starter library, but it interacts with the database by looking at the *application.properties*

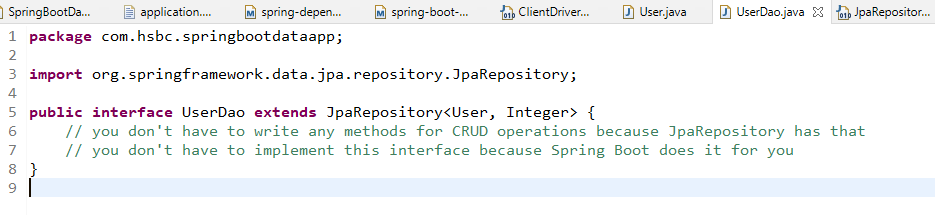
Spring Data Jpa

* You don’t have to write query
* You don’t have to write logic to establish connection or close connections
* You will get some predefined methods to perform all kind of tasks
* You will get the methods from the interface JpaRepository<T, ID>
* You don’t have to implement (class) DAO layer, spring data jpa does it for you
* You will write only dao interface that extends JpaRepository<T, ID>

User.java



UserDao.java



SpringBoot will provide implementations for all the methods of JpaRespository to perform on User entity.

Some of the methods of JpaRepository

findAll: List

save(T): T

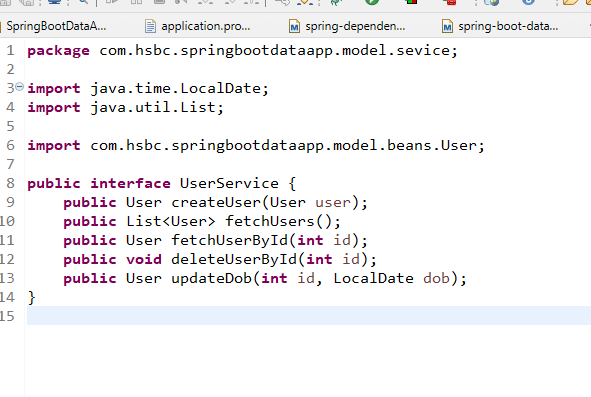
getOne(ID): T

deleteById(ID): void

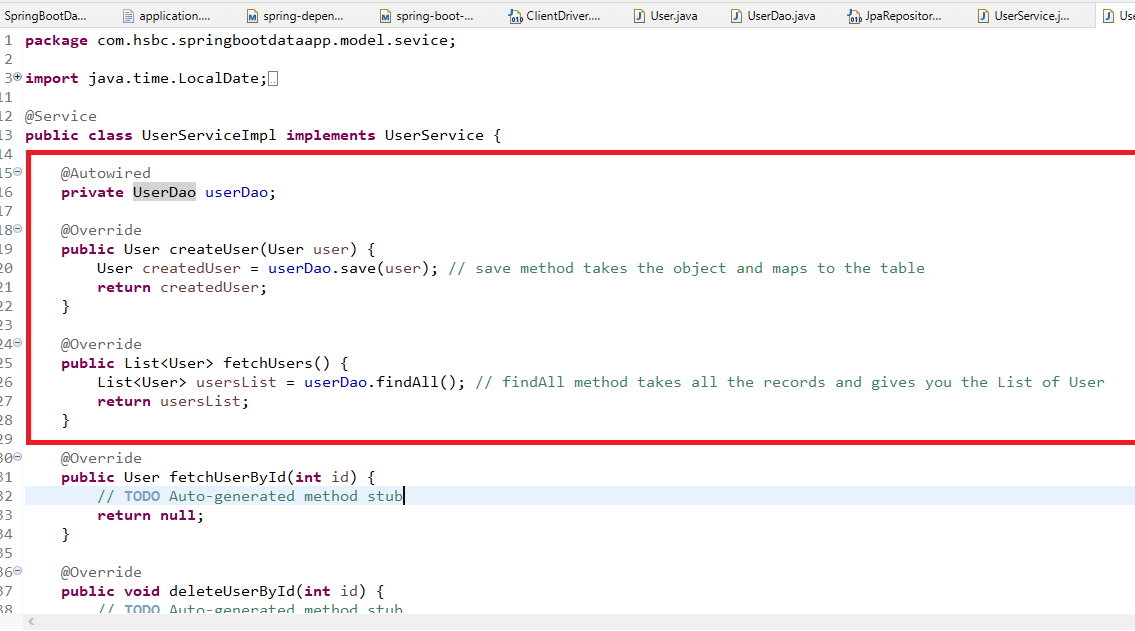
You don’t have to implement DAO, because spring boot will implement the methods present in DAO and it maintains the object of the DAO in the container so that you can use @Autowired and use the object in the service layer

We have to create only service & controller layer

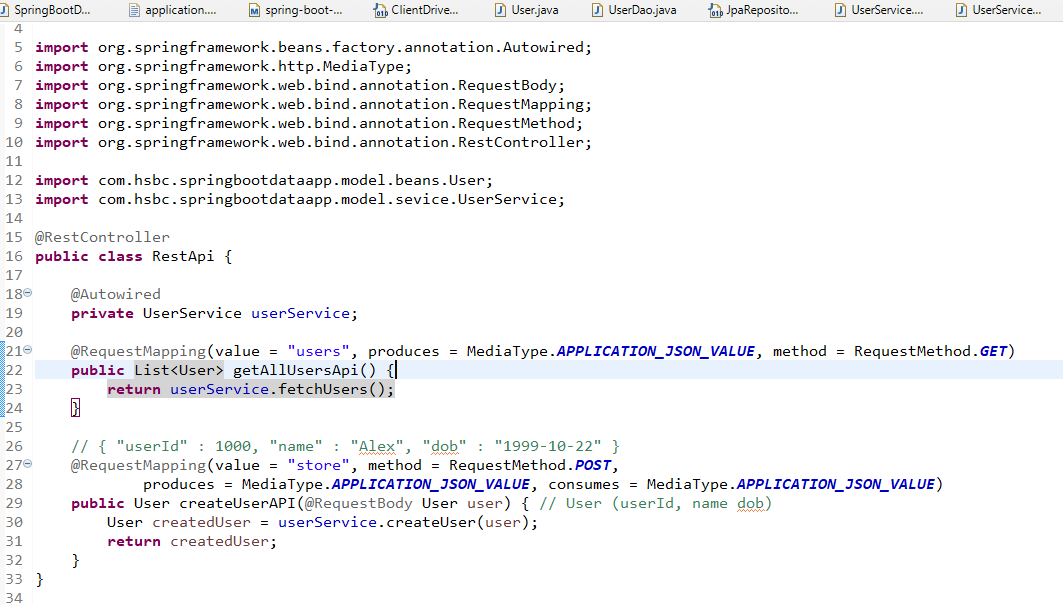
UserService.java



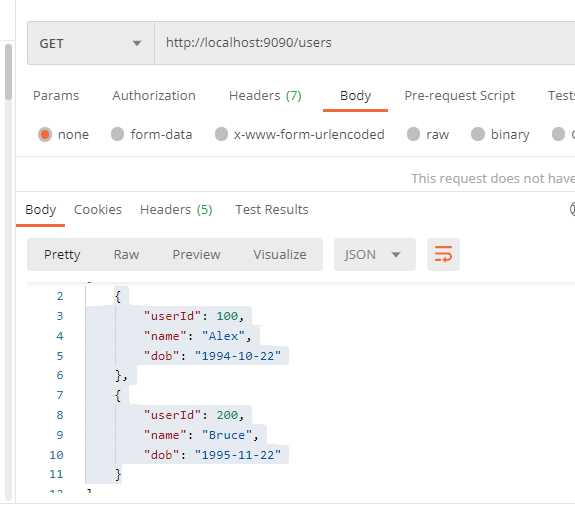
UserServiceImpl.java



RestApi.java



Output:



Exercise:

Implement other methods of Service layer by calling appropriate methods of DAO

Use different HTTP methods:

GET: fetching

POST: Storing

PUT: Updating

DELETE: Deleting

Download below softwares

* Node.js
* Visual Studio Code

Spring Microservices:

They are small services which are independently deployed on different servers so that if one service is down other services will be still available for the customers

Eureka Registry:

It is a common location where all the microservices are registered, even eureka registry also runs in a separate server.

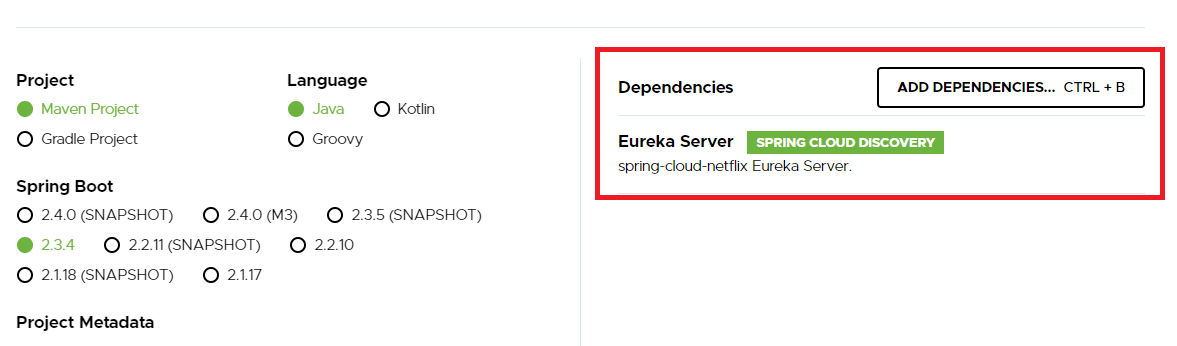
Spring Microservices will simplify the development & deployment of services by providing some starter libraries, it can be done only through spring boot projects

Some of the starter library

Eureka Server: Add eureka registry feature and register all your microservices

Eureka Client: A microservice that can be registered in the Eureka Server

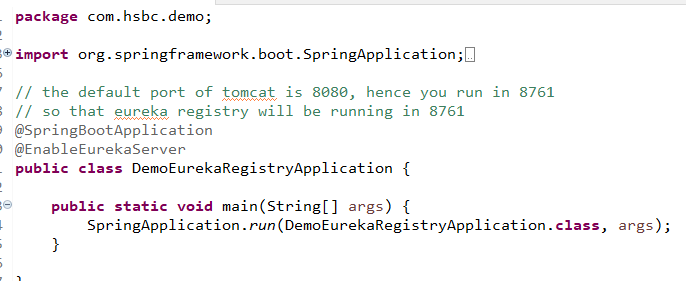
Eureka Registry:



@EnableEurekaServer: This annotation activates the application to act like a registry where all the microservices can register.

Note: By default microservices registers in eureka server in port 8761

Creating the registry



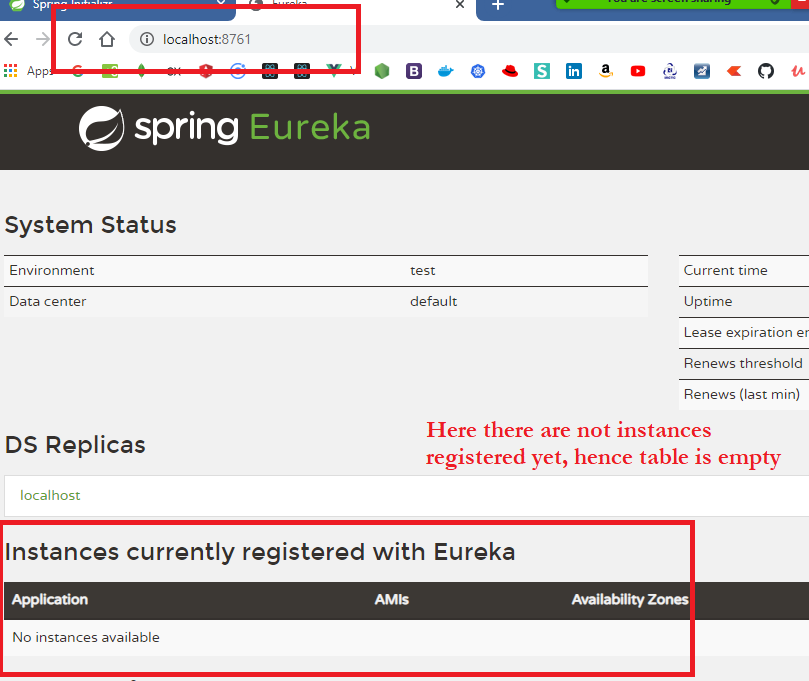
Changing the application.properties

server.port = 8761

eureka.client.register-with-eureka = false

eureka.client.fetch-registry = false

Note: By default eureka server also acts like a microservice & tries to register as a service to disable that you write above 2 properties



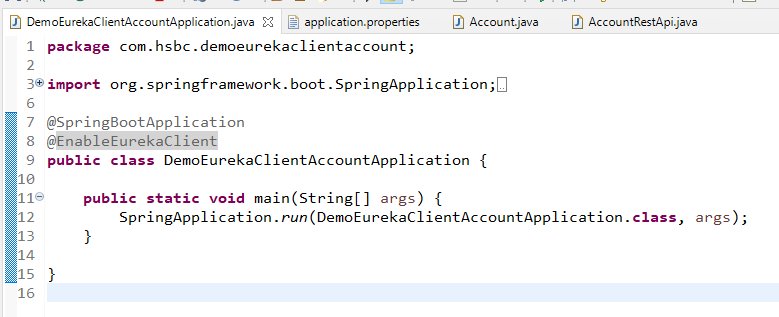
How to register the microservices

We need two dependencies

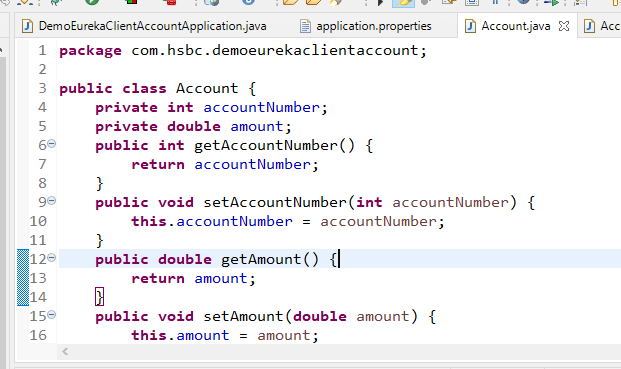
1. Eureka Client: @EnableEurekaClient
2. Web starter
3. Dev tools(Optional)



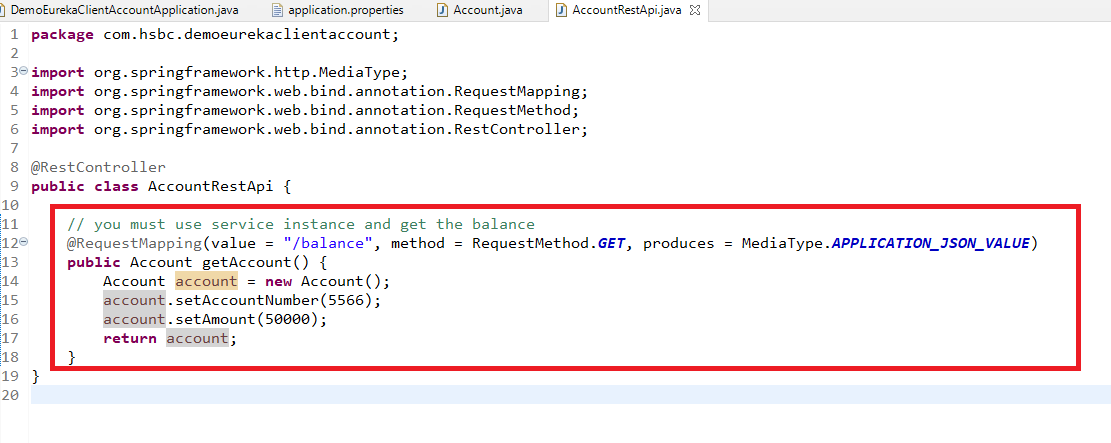
Eureka Client



Account.java



Independent Microservice: Which is a restful web service

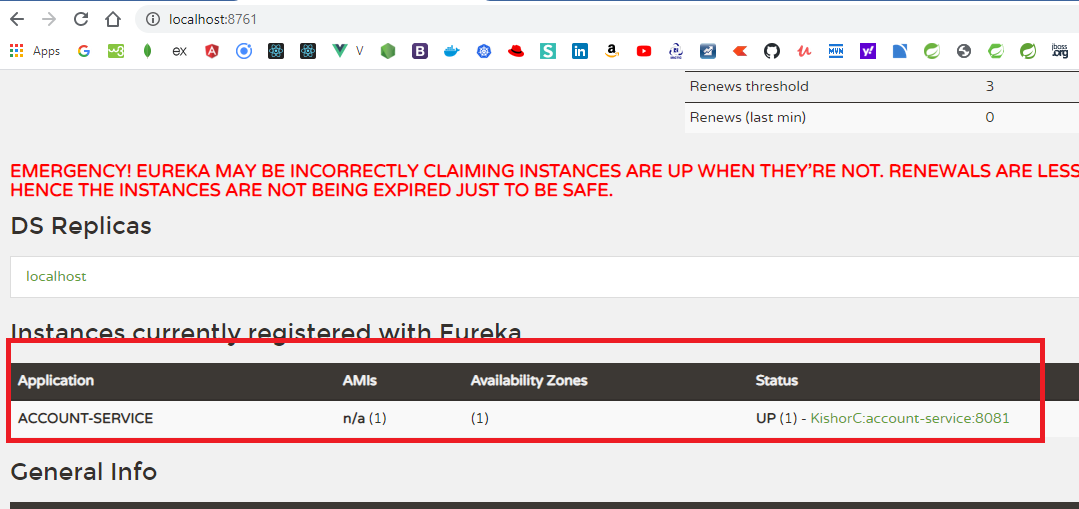


application.properties

server.port = 8081

spring.application.name = account-service

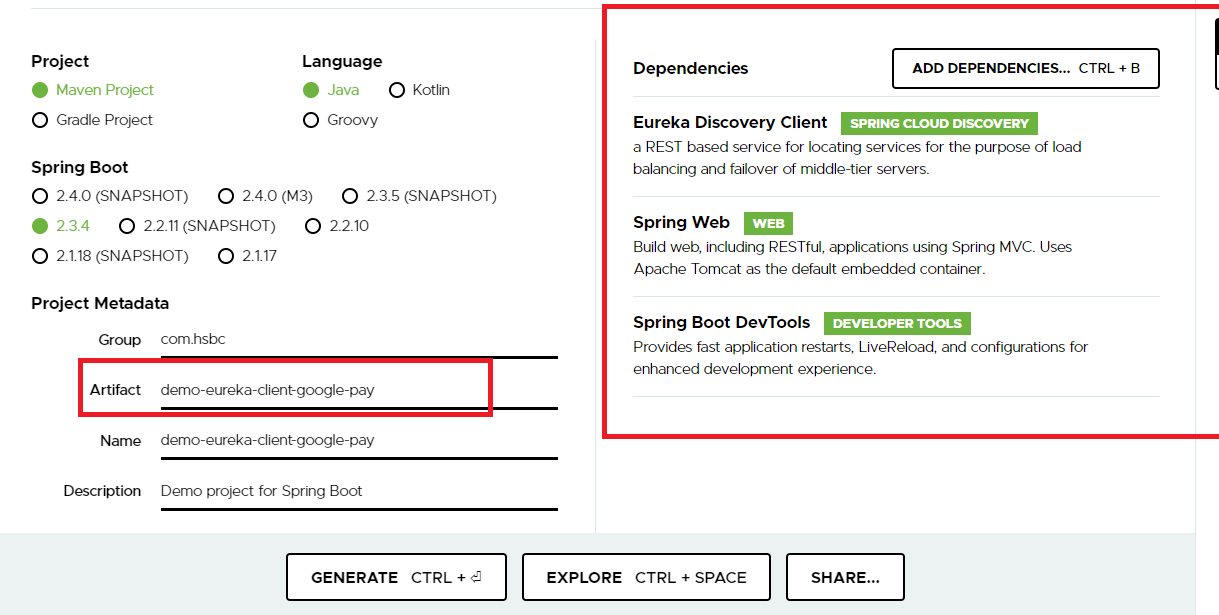
Once you run the account-service you can see this service in the eureka dashboard



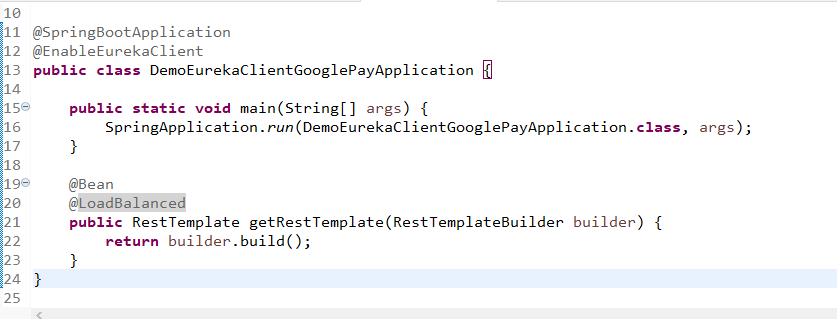
How to communicate from one microservice with another microservice

1. Microservices can communicate with another microservice via eureka registry
2. They use RestTemplate to call another microservice, it provides you a method where you can enter url of the microservice registered in the eureka
3. RestTemplate uses @LoadBalanced annotation to distribute the load

Let us create google pay service to call account service via Eureka



GooglePayService Main



RestApi of Google Pay

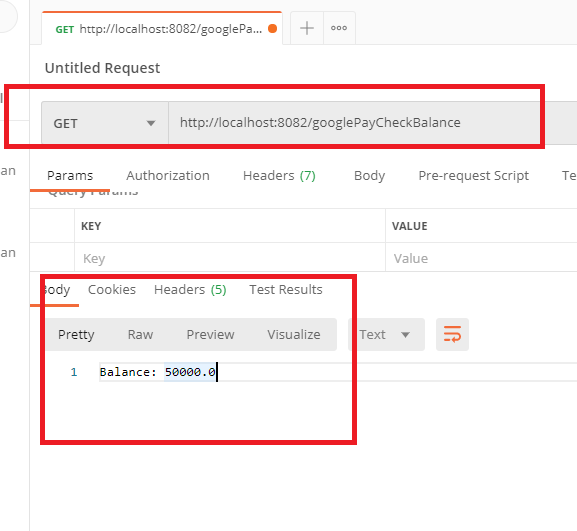


application.properties

server.port = 8082

spring.application.name = google-pay-service

Output:

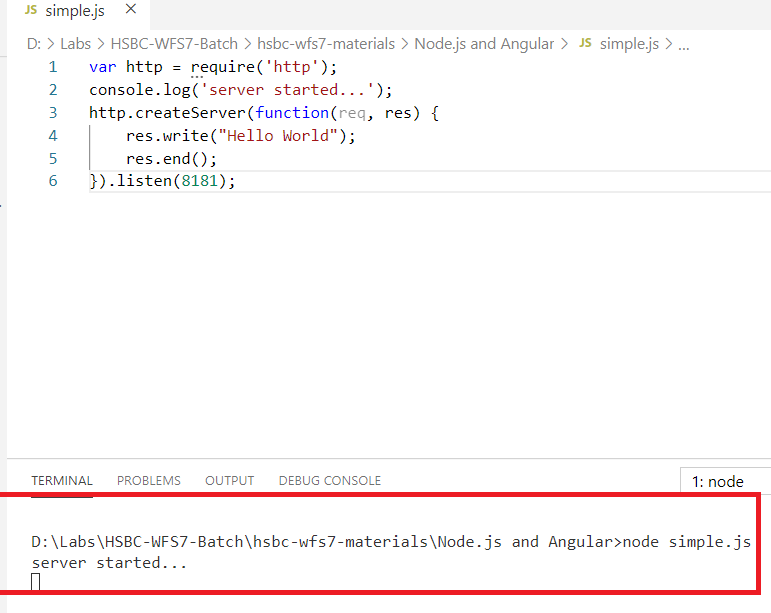


URL for Angular and Installation guides:

<https://www.youtube.com/channel/UCFlwQKoyB0Qo_y1B3JNl10g/playlists>

Node.js & Angular

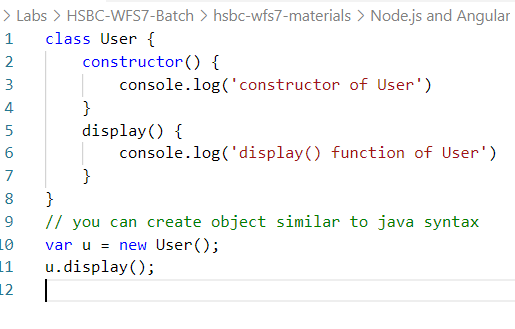
Node.js is a javascript runtime environment which allows you to run your javascripts at the backend ie., server side.



Node.js provides lot of inbuilt libraries using that you can create complex applications using javascript, it also provides a command `npm` stands for Node Package Manager which is used download some *external tools & libraries* required for your javascript program

Javascript has many new features (ES6 Features) that made javascript simpler to write

* let, const keywords
* class, constructor & super keywords
* template strings
* arrow functions

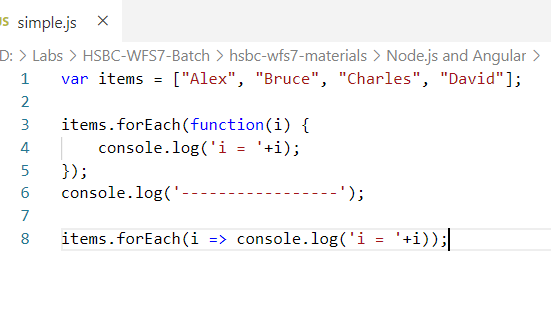


Template Strings



Arrow Functions (=>):

It simplifies writing anonymous functions in the javascript, it is similar to lambda expression of Java



Typescript:

It is a super to the javascript it adds types to the javascript so that data will get right type value

function add(x, y) {  
 return x + y  
}

add function can take any kind of value

add(10, “abc”)  
add(“abc”, “xyz”);  
add(true, false);

To avoid wrong type values to pass you can use typescript that can be converted to javascript with the transpiler (compiler of javascript)

function add(x : number, y : number) {   
 return x + y;  
}

add(20, 30); // this compiles

add(“abc”, “xyz”); // typescript compiler raises error.

Typescript has many types:

number  
string  
boolean  
void  
Date  
any

All these are the datatypes provided typescript

abc.ts (typescript file) -> abc.js (javascript file)

Angular Framework:  
It is used to develop single page applications using HTML and Typescript language

HTML is used for building User Interfaces

Typescript is used for writing business logics.

Angular is used to develop web & mobile applications

Angular Framework:

It provides

* Angular CLI tool kit
  + You can create ready to run angular applications
  + You will get embedded server to host angular applications
  + you will get typescript compiler to compile typescript to javascript
  + It auto-compiles your application
  + You will get auto-reload features while you are writing the code

Angular CLI toolkit is installed using npm command

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

Once you install angular cli you can use the command `ng`

Using ng command you can

1. create project
2. host project
3. generate some additional features required to angular project

To verify angular CLI toolkit

*`ng --version`*

You can create angular project using `*ng new app-name`*

You can run angular project using `ng serve`

\*ngFor: it is used to iterate the array, it is angular structural directive

To generate the component we have a command

*`ng g c component-name`*

If you use above command it creates html, ts, css and also declares the component in the AppModule

Pipes:

Angular provides pipes to format the output, we have many inbuilt pipes

date  
uppercase  
lowercase  
currency  
json

{{variable | uppercase }}

in mock.ts add employee array having id, name, salary and dob and show the employee data in a display-employee component in a table format

|  |  |  |  |
| --- | --- | --- | --- |
| Employee Id | Name | DOB(dd/MM/yyyy) | Salary in INR |
| .. | .. | .. | .. |
| ... | ... | ... | ... |
|  |  |  |  |

Databinding:

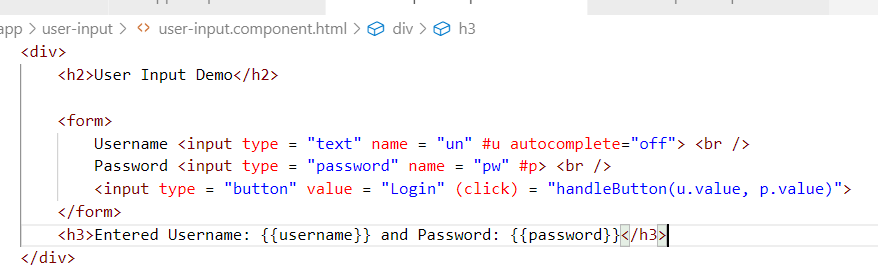
It allows you to bind the data between view and the component class, you have 4 types of databinding

1. Interpolation {{property}}: Component -> View (One way binding)
2. Event binding (event): View -> Component (One way binding)
3. Property binding [property]: Component -> View (One way binding)

It is mainly used to manipulate the DOM properties, like hiding the element, disabling button

1. Two way databinding [(ngModel)]: Component <-> View

Event Binding:



user-input.component.ts



Property binding []:

It is used to manipulate the dome elements like disabling, enabling buttons, hiding/showing the elements



user-input.component.ts

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

@Component({

  selector: 'app-user-input',

  templateUrl: './user-input.component.html',

  styleUrls: ['./user-input.component.css']

})

export class UserInputComponent {

  username  = undefined;

  password = undefined;

  isDisabled = true;

  handleButton(u, p) {

    this.username = u;

    this.password = p;

    console.log('handleButton() called');

  }

  handleKeyInput(u) {

    console.log('handleInput() called: '+u)

  }

  enableButton(p:string) {

    if(p.length >= 3) {

      this.isDisabled = false;

    } else {

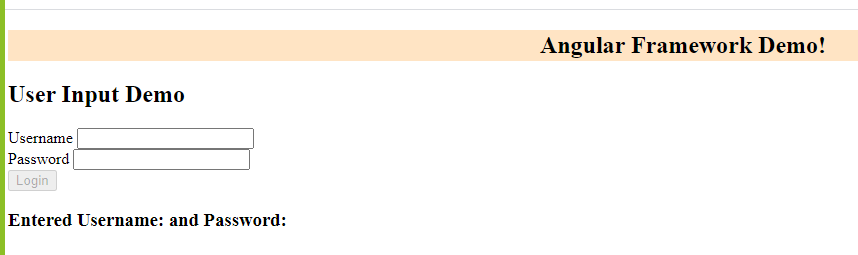
      this.isDisabled = true;

    }

  }

}

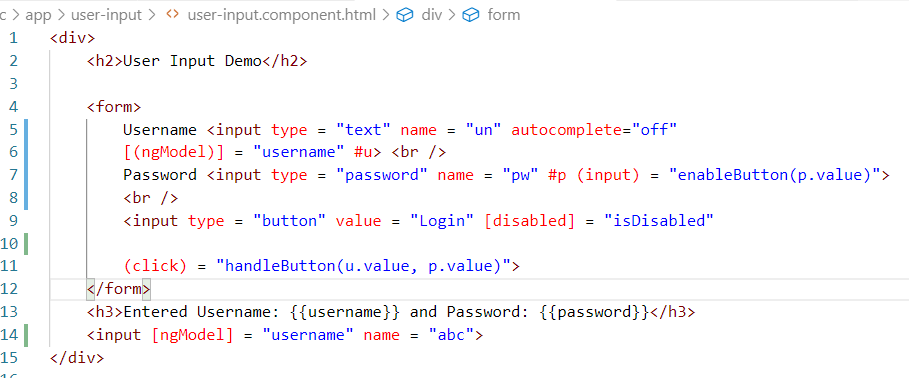
Output:



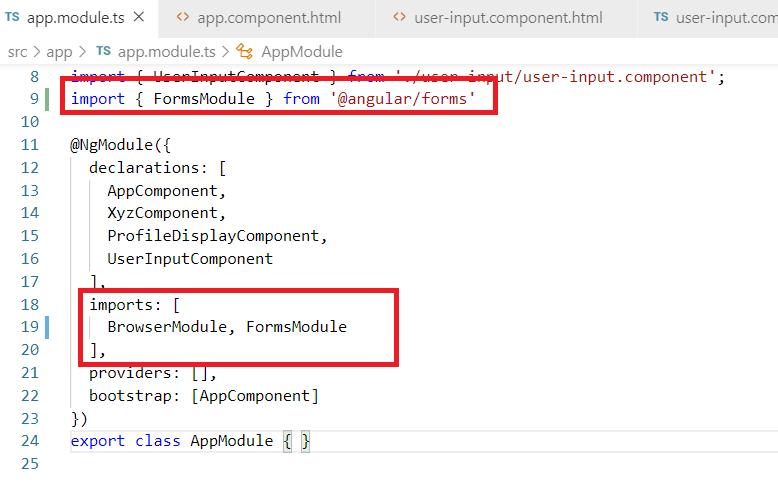
Two way databinding: combines property & event binding both using which you can see data passing in both directions.

[(ngModel)] = “property”

ngModel is part of one sub-module called FormsModule, you must add this module in the *AppModule imports* attribute



Update the AppModule to add FormsModule from *@angular/forms*



*@Input & @Output* decorators to share data between the components i.e., from parent to child and child to parent

@Input() is used when parent component data to be passed to child component

@Output() is used when child component data to be passed to parent component

parent.component.html

<div>

    <div style = "text-align: center;">

        <h2>This is parent component</h2>

        <input type = "text" name = "pn" [(ngModel)] = "parentName">

        <h2>Parent Component name : {{parentName}}</h2>

    </div>

    <hr />

    <app-child [childName] = "parentName"></app-child>

    <hr />

    <div style = "text-align: center;">

        <app-names-item [username] = "names[2]"></app-names-item>

    </div>

    <hr />

    <div>

        <ul>

            <li \*ngFor = "let n of names">

                <app-names-item [username] = "n"></app-names-item>

            </li>

        </ul>

    </div>

</div>

child.component.html

<div>

    <h2>This is child component</h2>

    <h2>Child Component name {{childName}}</h2>

</div>

names-item.component.html

<div>

    <h3>Hello {{username}} !</h3>

</div>

parent.component.ts

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

@Component({

  selector: 'app-parent',

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

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

})

export class ParentComponent {

  parentName = "Alexandar";

  names = ["Alex", "Bruce", "Chandler", "David"];

}

child.component.ts

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

@Component({

  selector: 'app-child',

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

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

})

export class ChildComponent  {

  @Input()

  childName = undefined;

}

names-item.component.ts

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

@Component({

  selector: 'app-names-item',

  templateUrl: './names-item.component.html',

  styleUrls: ['./names-item.component.css']

})

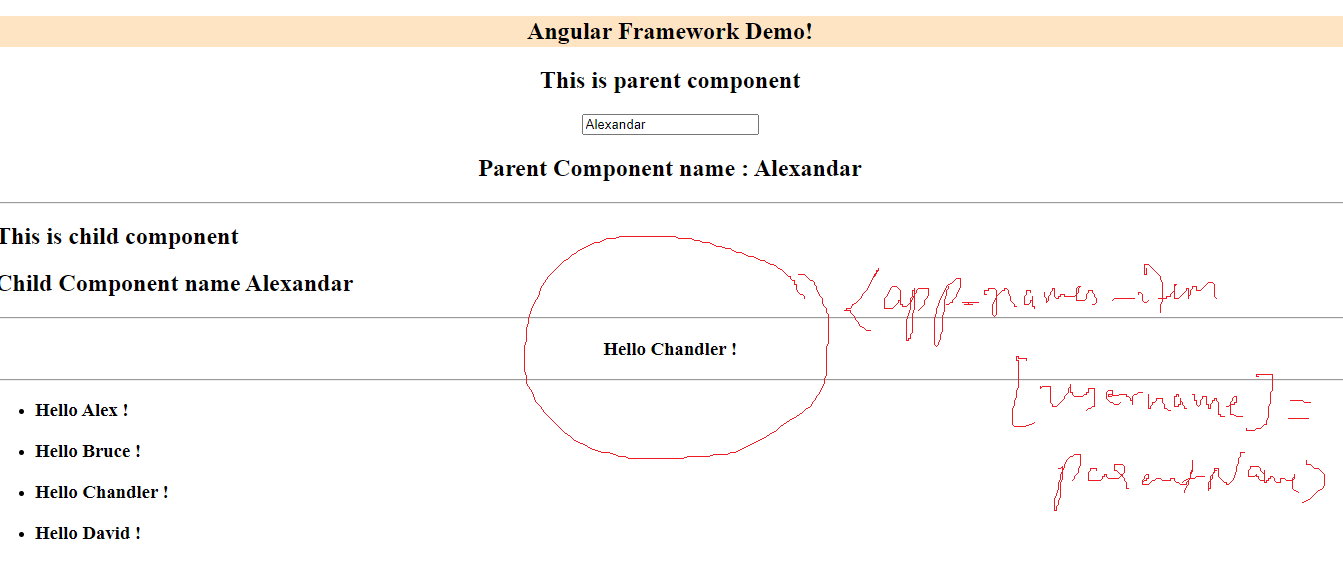
export class NamesItemComponent  {

  @Input()

  username = "Zaheer";

}

Output:



@Output: It is used to share the data from child component to parent component

child.component.ts



child.component.html

<div>

    <h2>This is child component</h2>

    <h2>Child Component name {{childName}}</h2>

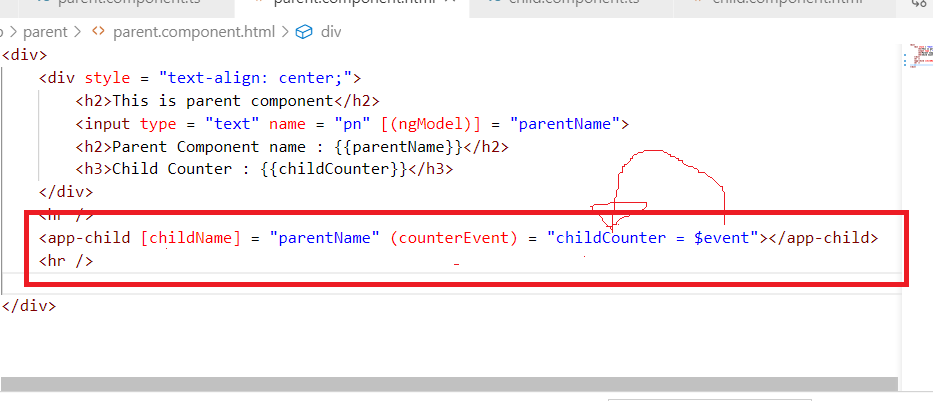
    <button (click) = "incrementCounter()">Increment</button>

</div>

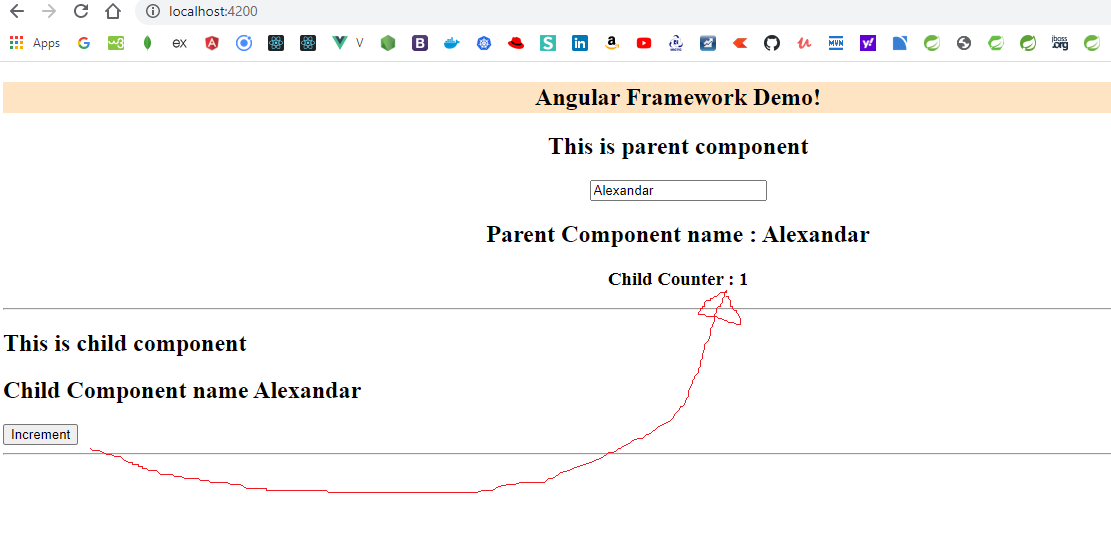
parent.component.ts



parent.component.html



Output:



Exercise:

Parent Component

Likes:

Dislikes:

Child Component

Like Button and Dislike Button

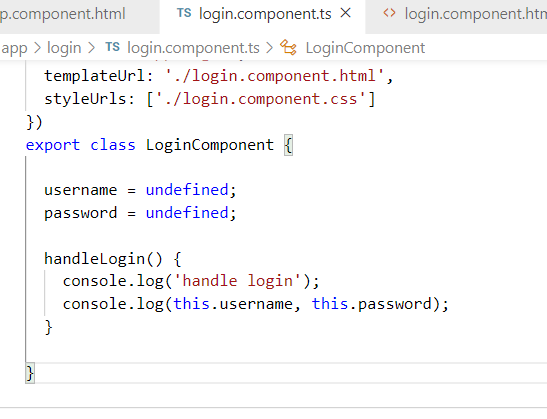
Angular Forms:

It allows you to submit the user input to the application

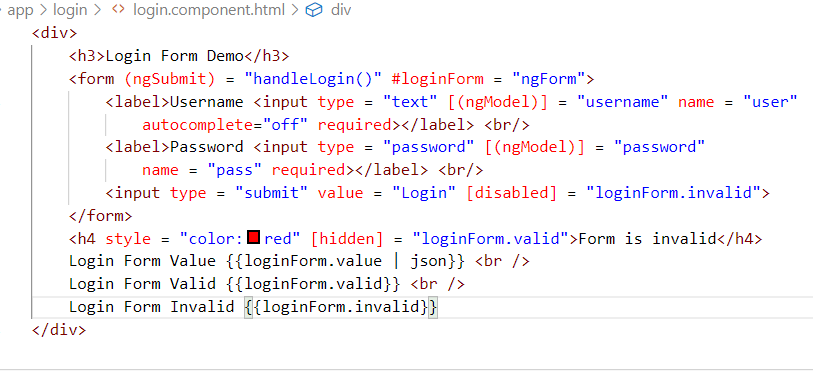
Login Form



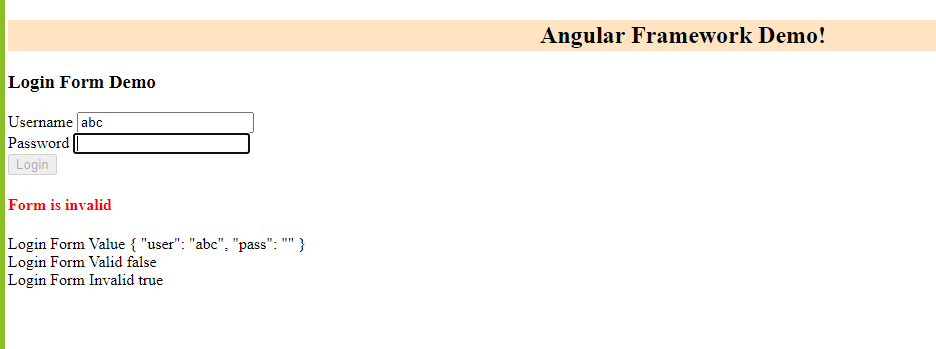
Login Form class



The above code is not validating the form



Output:



You can add bootstrap css and use some of the inbuilt classes

Note: ngModel is used in the below form to access each input fields properties like pristine, valid, invalid

pristine: it will be true if control is not modified

<div class = "container-fluid">

    <h3>Login Form Demo</h3>

    <div class = "form-group">

        <form #login = "ngForm">

            <div class = "row">

                <div class = "col-3">

                    <input type = "text" name = "user" autocomplete="off"  #un = "ngModel"

                       [(ngModel)] = "username" class = "form-control" placeholder="Enter Username" required>

                </div>

                <div class = "col-3">

                    <input type = "password" name = "pass" #pw = "ngModel"

                       [(ngModel)] = "password" class = "form-control" placeholder="Enter Password" required>

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <div class = "alert alert-danger" [hidden] = "un.valid||un.pristine">Username is required</div>

                </div>

                <div class = "col-3">

                    <div class = "alert alert-danger" [hidden] = "pw.valid||pw.pristine">Password is required</div>

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <input type = "submit" value = "Login" class = "btn btn-primary">

                </div>

            </div>

        </form>

        <div>

            <h2>Understandign of ngModel properties</h2>

            <h3>un ngModel</h3>

            <p>Un Valid: {{un.valid}}</p>

            <p>Un InValid: {{un.invalid}}</p>

            <p>Un Value: {{un.value}}</p>

            <p>Un Pristine: {{un.pristine}}</p>

            <h3>pw ngModel</h3>

            <p>Pw Valid: {{pw.valid}}</p>

            <p>Pw InValid: {{pw.invalid}}</p>

            <p>Pw Value: {{pw.value}}</p>

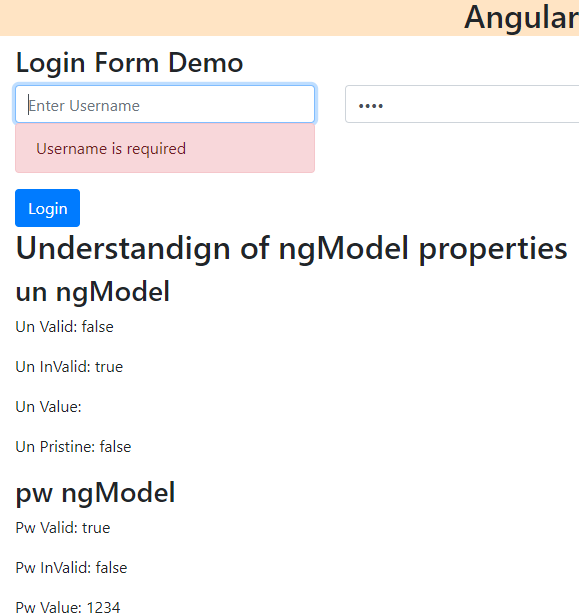
            <p>Pw Pristine: {{pw.pristine}}</p>

        </div>

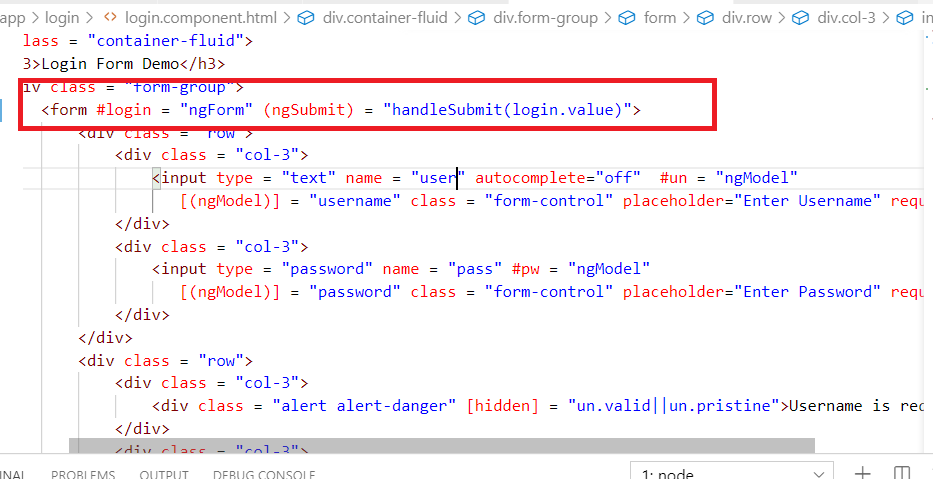
    </div>

</div>

Output:



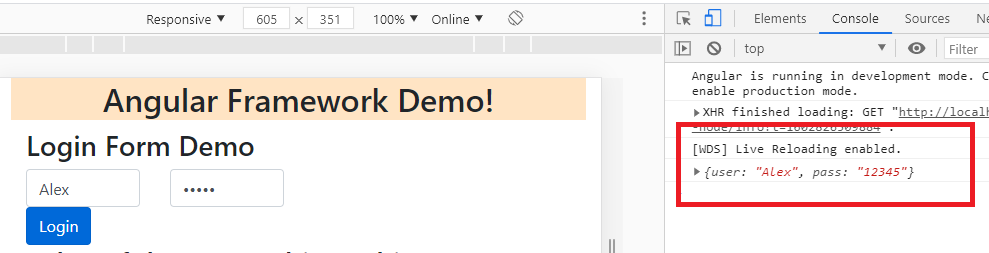
We also can send ngForm objects directly to the components using its value property



login.component.ts



Output:



Notice in the console you get javascript object which has two properties *user & pass*, which are name of the *<input>*

Angular has some basic building blocks

1. Modules: represents whole application or part of the application
2. Components: reusable user interfaces
3. Pipes: Transforming the output
4. Databinding: To pass the data
5. Services: Reusable logics called from different components
6. Router: Navigating from one component to another component

Angular Services

Services are reusable logics that multiple components can call, a service is a class with *@Injectable()* decorator

@Injectable() makes angular framework to instantiate the service and supply to the components in the form dependency injection (constructor dependency injection)

Service:   
@Injectable()  
export class AppService { ... }  
  
Component:

@Component({...})  
export class AppComponent {   
 constructor(private AppService service) { }   
}

Note: Typescript will use one short cut, i.e, if your constructor parameter is private then no need of writing initializing the variable

i.e.,  
username = undefined; password = undefined;

constructor(username: string, password : string) {   
 this.username = username;  
 this.password = password;  
}

Short-cut initialization:

constructor(private username: string, private password:string) { }

How to generate angular service through commands

*ng generate service service-name (or)*

*ng g s service-name*

app-service.service.ts

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

import { USER\_ARRAY } from './mock';

@Injectable({

  providedIn: 'root'

})

export class AppServiceService {

  constructor() { }

  login(username : string, password : string) : void {

    if(username == 'Alex') {

      console.log('log in successful')

    } else {

      console.log('login failed')

    }

  }

  fetchUsers() : any {

    // import USER\_ARRAY from mock.ts

    return USER\_ARRAY;

  }

}

app.component.html

<div class = "container-fluid">

  <h2 class = "heading">Angular Framework Demo!</h2>

  <!-- <app-profile-display></app-profile-display> -->

  <!-- <app-user-input></app-user-input> -->

  <!-- <app-parent></app-parent> -->

  <div class = "row">

   <div class = "col-6">

    <app-profile-display></app-profile-display>

   </div>

   <div class = "col-6">

    <app-login></app-login>

   </div>

  </div>

</div>

profile-display.component.ts

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

import {AppServiceService} from '../app-service.service'

@Component({

  selector: 'app-profile-display',

  templateUrl: './profile-display.component.html',

  styleUrls: ['./profile-display.component.css']

})

export class ProfileDisplayComponent  {

  userCollection : any = undefined;

  constructor(private service: AppServiceService) {}

  handleClick() {

    this.userCollection = this.service.fetchUsers();

  }

}

profile-display.component.html

<div>

    <h2>Profile Display Component</h2>

    <button (click) = "handleClick()" class = "btn btn-primary">Show Profiles</button>

    <table class = "table">

        <tr>

            <th>Name</th>

            <th>Dob</th>

        </tr>

        <tr \*ngFor = "let user of userCollection">

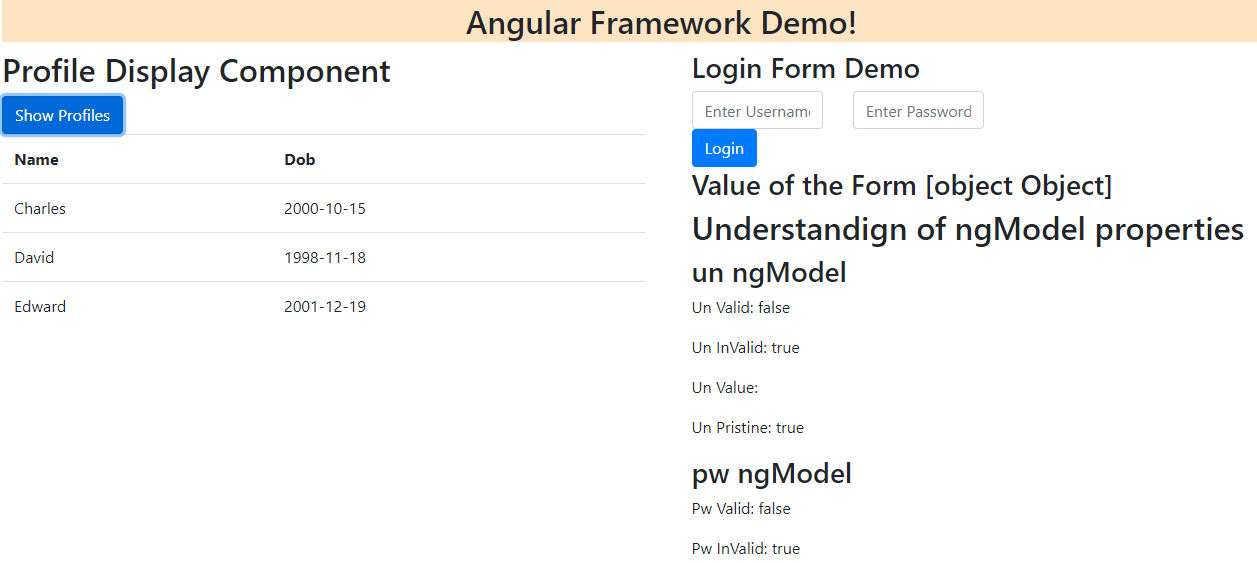
            <td>{{user.name}}</td><td>{{user.dob}}</td>

        </tr>

    </table>

</div>

Output;



HttpClient

It is used to call the external Webservice from angular, it is part of a submodule called *HttpClientModule*. This module you had to mention in the AppModule so that angular creates HttpClient instance, using HttpClient you can call webservices through http methods like get, post, put and delete.

HttpClient has http methods:

get(url) : Observable<T>  
post(url, data) : Observable<T>  
put(url, data) : Observable<T>  
delete(url, data): Observable<T>

Observable: maintains the response coming from the Webservice you can extract the response data from Observable with the help ‘subscribe()’ method

HttpClient instance must be used in Angular Service

Note: HttpClient instance will be available to the service through the dependency injection, because angular passes this object when you mention HttpClientModule in the AppModule

Adding HttpClientModule in the AppModule `@NgModule imports`, its part of @angular/common/http



Use HttpClient instance in the Angular Service via constructor dependency injection

app.service.service.ts



Now the Webservice must allow the request coming from the cross-origin hence you will modify the spring Webservice to accept request from angular application

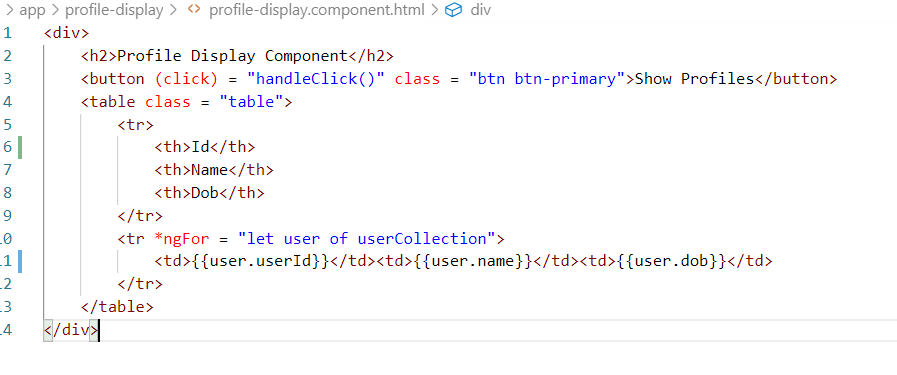
Java Webservice code



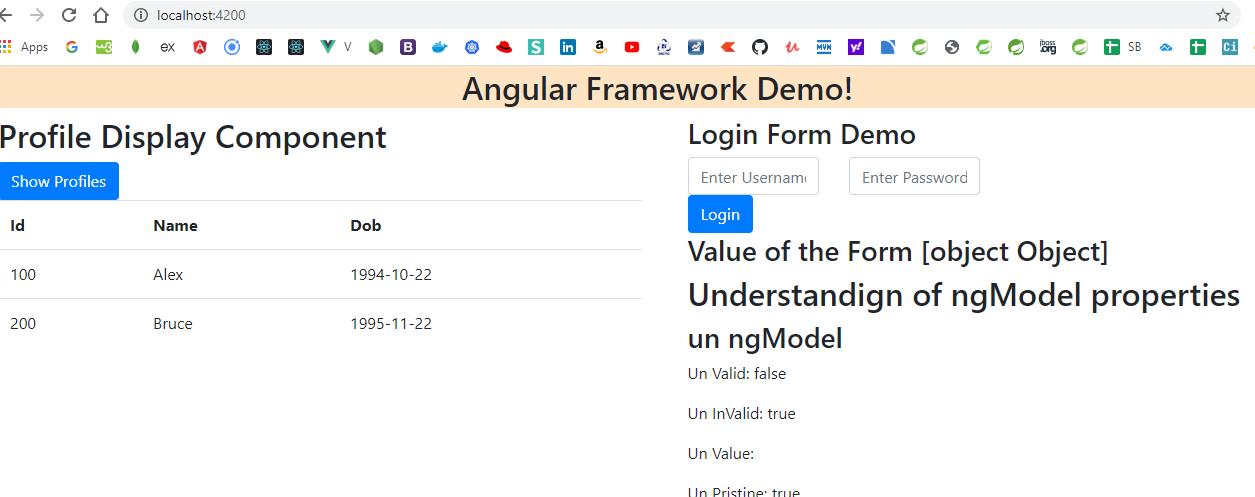
profile-display.component.ts



profile-display.component.html



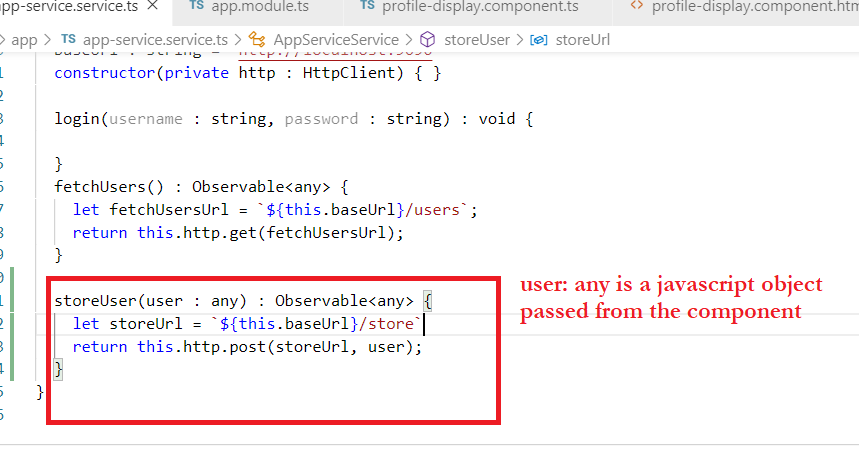
Output:



Sending the form data and calling post Webservice

You must call post(url, data) of HttpClient

Creating a method in a Service class



user-registration.html

<div class = "container-fluid">

    <h3>Registration Form</h3>

    <hr />

    <h4 \*ngIf = "message">Status: {{message}}</h4>

    <div class = "form-group">

        <form #registration = "ngForm" (ngSubmit) = "handleStore(registration.value)">

            <div class = "row">

                <div class = "col-3">

                    <label>Enter User Id</label>

                </div>

                <div class = "col-4">

                    <input type = "text" name = "userId" [(ngModel)]= "userId" class = "form-control">

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <label>Enter Name</label>

                </div>

                <div class = "col-4">

                    <input type = "text" name = "name" [(ngModel)]= "name" class = "form-control">

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <label>Birthday:</label>

                </div>

                <div class = "col-4">

                    <input type = "date" name = "dob" [(ngModel)]= "dob" class = "form-control">

                </div>

            </div>

            <div class = "row">

                <input type = "submit" value = "Register" class = "alert alert-primary">

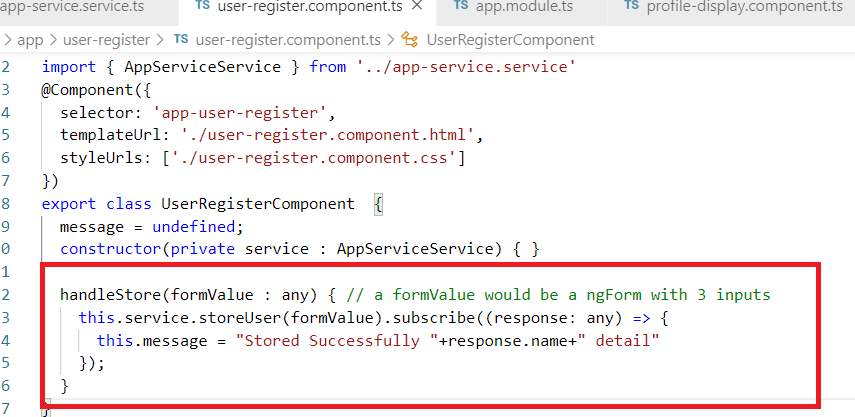
            </div>

        </form>

    </div>

</div>

user-register.ts



Output:



Angular Router

It enables you to load the components when you click on a link or a button, so that you can load multiple components in the same page, inorder to achieve this angular needs a sub-module called RouterModule which must be mentioned in the AppModule imports section.

@NgModule({  
 ...  
 imports = [.., RouterModule.forRoot(routes)...]  
 ...  
})  
export class AppModule { }

The forRoot method takes the routes which has path & component configurations

i.e., when you click on a link some path will be generated for that path what component should be shown on a page.

<router-outlet>: It is a placeholder to load the components based on the routes

Routes: it configurations the path & components

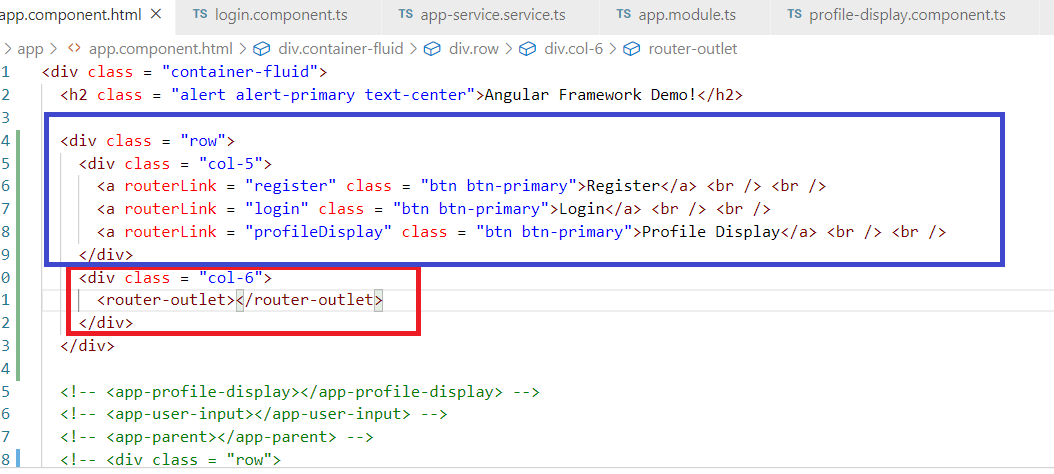
RouterModule.forRoot(): it takes the routes and load the components to the router-outlet based on the paths.

Configuring the RouterModule

app.module.ts



Adding <router-outlet> in the app component



Output:



Router: It is used to programmatically navigate by providing the url in the navigate method, Router instance can be provided through constructor-injection.

Nested Routers

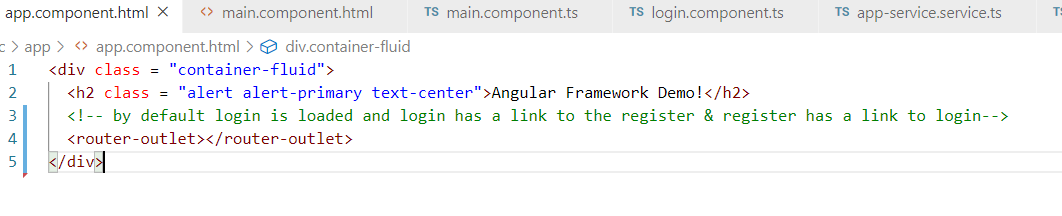
When you have a component that is loaded in a router it might have another router where child components could be loaded.

In the app.component.html we will load three components

1. login
2. register
3. success

Login authenticates the user and loads success else loads login back, here you need programmatic navigation i.e, Router instances that you can get through constructor injection.

app.component.html



app.module.ts



register.component.html

<div class = "container-fluid">

    <h3>Registration Form</h3>

    <hr />

    <h4 \*ngIf = "message">Status: {{message}}</h4>

    <div class = "form-group">

        <form #registration = "ngForm" (ngSubmit) = "handleStore(registration.value)">

            <div class = "row">

                <div class = "col-3">

                    <label>Enter User Id</label>

                </div>

                <div class = "col-4">

                    <input type = "text" name = "userId" [(ngModel)]= "userId" class = "form-control">

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <label>Enter Name</label>

                </div>

                <div class = "col-4">

                    <input type = "text" name = "name" [(ngModel)]= "name" class = "form-control">

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <label>Birthday:</label>

                </div>

                <div class = "col-4">

                    <input type = "date" name = "dob" [(ngModel)]= "dob" class = "form-control">

                </div>

            </div>

            <div class = "row">

                <input type = "submit" value = "Register" class = "alert alert-primary">

            </div>

        </form>

        <br />

<!-- the link that takes you to login component-->

        <a routerLink = "/login">Login</a>

    </div>

</div>

login.html

<div class = "container-fluid">

    <h3>Login Form Demo</h3>

    <div class = "form-group">

        <form #login = "ngForm" (ngSubmit) = "handleSubmit(login.value)">

            <div class = "row">

                <div class = "col-3">

                    <input type = "text" name = "user" autocomplete="off"  #un = "ngModel"

                       [(ngModel)] = "username" class = "form-control" placeholder="Enter Username" required>

                </div>

                <div class = "col-3">

                    <input type = "password" name = "pass" #pw = "ngModel"

                       [(ngModel)] = "password" class = "form-control" placeholder="Enter Password" required>

                </div>

            </div>

            <div class = "row">

                <div class = "col-3">

                    <div class = "alert alert-danger" [hidden] = "un.valid||un.pristine">Username is required</div>

                </div>

                <div class = "col-3">

                    <div class = "alert alert-danger" [hidden] = "pw.valid||pw.pristine">Password is required</div>

                </div>

            </div>

            <br />

            <div class = "row">

                <div class = "col-3">

                    <input type = "submit" value = "Login" class = "btn btn-primary">

                </div>

            </div>

        </form>

        <br />

        <!-- the link that takes you to register-->

        <a routerLink = "/register">Create User</a>

    </div>

</div>

login.component.ts

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

import { Router } from '@angular/router';

@Component({

  selector: 'app-login',

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

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

})

export class LoginComponent {

  username = undefined;

  password = undefined;

  // supply the router instance through constructor injection

  constructor(private \_router : Router) {}

  handleSubmit(formValues : any) {

    //you must call the service method to authenticate

    let user = formValues.user;

    let pass = formValues.pass;

    if(user == 'Admin' && pass == '1234') {

      this.\_router.navigate(["/success"]);

    } else {

      this.username = undefined;

      this.password = undefined;

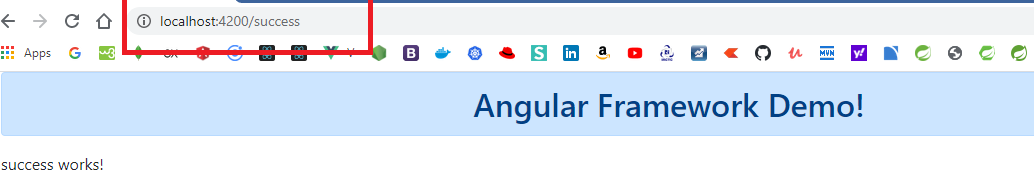
      this.\_router.navigate(["/login"]);

    }

  }

}

Output:

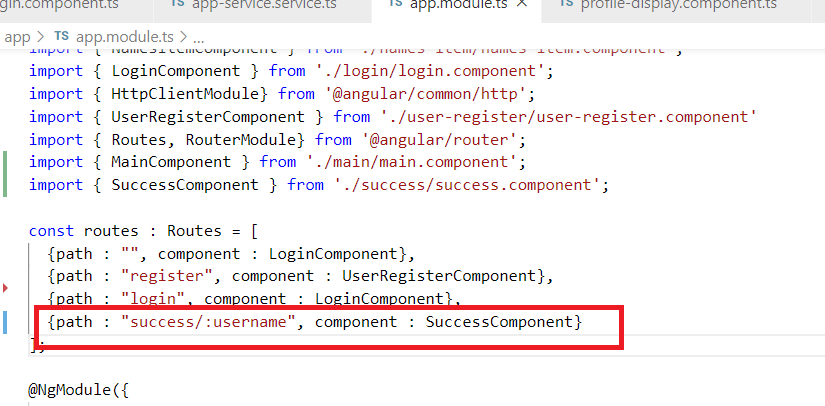


Here success works is loaded based on /success url but you may want to load the success based on the username so you need to change the path to take a dynamic url like

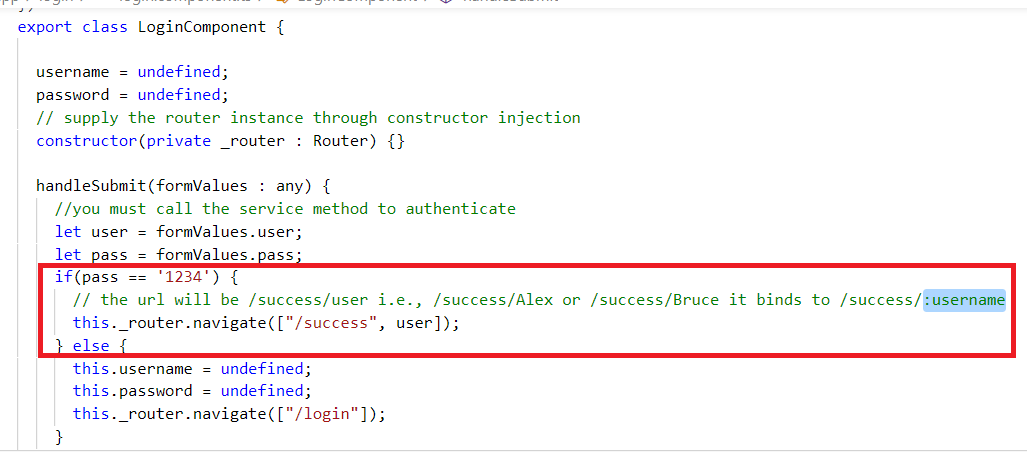
*/success/:username*

Where :username is a binding variable which can accept any value in the url

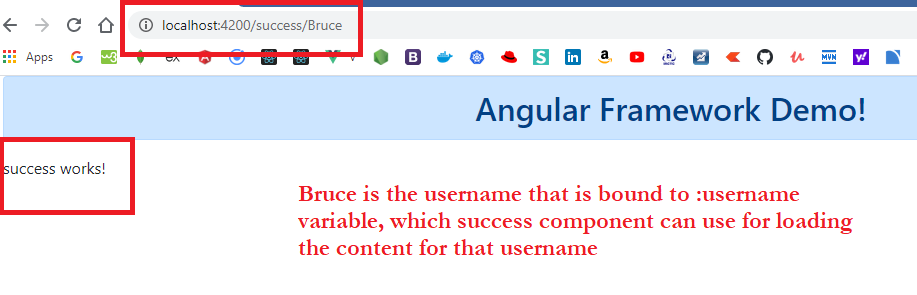
Changes in app.module.ts



login.component.ts must load to the /success/:username as below



Output:

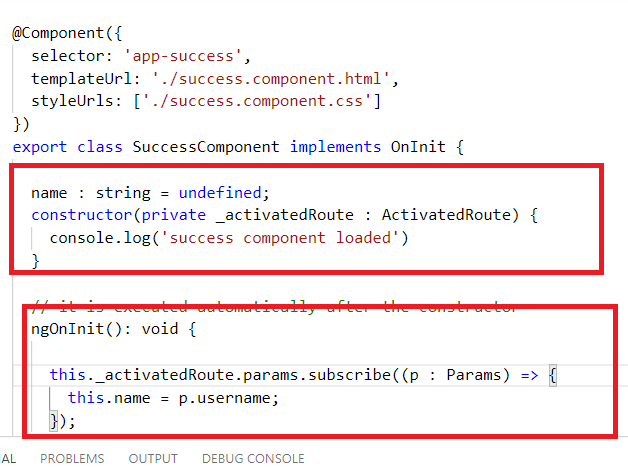


Now you can have another instance to extract the values from the URL

ActivatedRoute: It is an instance which extracts the values from the path of the application

You can get the instance of ActivatedRoute through constructor injection

success.component.ts



success.component.html



app.module.ts



dashboard.component

<p>dashboard works!</p>

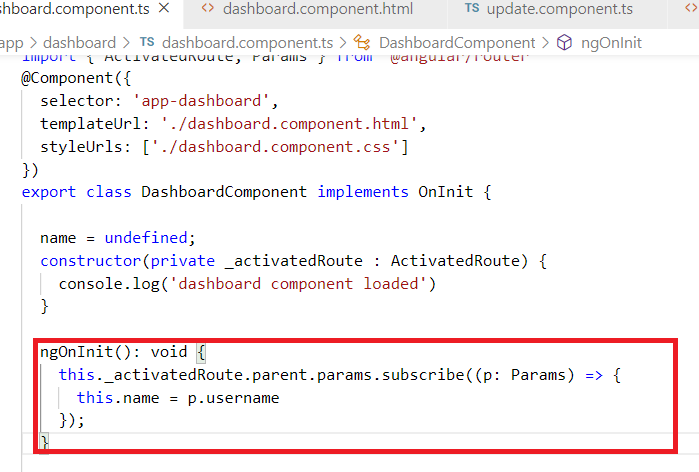
update.component

<p>update works!</p>

Output:



How to extract the values from the parent path



Summary

* Spring Framework uses eager loading by default and to enable lazy loading you must use *lazy-init = true* in the <bean>
* Spring Framework creates singleton instance by default to enable multiple instance you need *scope = prototype* to be used
* Annotations that let spring to create the object
  + @Component
  + @Service
  + @Repository
  + @Controller
  + @RestController
* Spring MVC
  + ModelAndView: It is an instance that specifies the view & the model where it specifies what jsp to be shown and it should show what model
* ${}: Expression Language, it is used to display the model
* Spring Boot:
  + Spring Data Jpa Starter library configures the database setup