Contents

* Spring Data JPA
* Angular
* Spring REST
* Spring Microservices
* AWS

Spring Data JPA

Pre-requisites:

* SQL
* JDBC
* ORM/Hibernate/JPA
* Spring Boot

Drawbacks of JDBC

1. You need to write code to create the connection instance
2. You don’t follow any standards to maintain the datasource informations, because you can keep the datasource informations either in Java file/Properties/XML and so on.
3. Most of the checked exceptions should be handled for every operation’s
4. SQL queries are database dependent, the sql queries you write in the JDBC may not work in all the database
5. You write too much code, only to perform type conversion from Java to SQL and vice versa
6. Lot of boilerplate code are written in JDBC.
7. Queries will become more complex if the operations are complex, ex: join queries, sub-queries

ORM (Object Relational Mapping) Framework:

This overcomes the drawback of JDBC, ORM framework will map java objects to the database tables with some predefined functions provided by ORM framework, it avoids lot of things like

1. Type conversion is not required i.e., Java to SQL & vice versa
2. Follows the standard, i.e., datasource informations must be kept in the configuration file (XML)
3. Queries are database independent and in ORM you can avoid writing queries
4. It can create connection for you which you can use it
5. You don’t have to handle any checked exceptions
6. Boiler plate codes are avoided
7. Complex operations can be done with simple code like complex joins can be done with simple annotations

ORM is a specification of JPA (Java Persistence API) i.e., standard, there are many organizations provided the implementation to the ORM

Some of the ORM frameworks are:

1. Hibernate (red-hat)
2. JPA (sun microsystem)
3. Toplink
4. iBatis

Out of the above ORM frameworks, hibernate is the one which is more widely used

Interacting with the database using Hibernate

Steps:

1. you need to create maven project
2. you need to add the hibernate dependencies
3. you need to add the jdbc driver dependencies
4. you need to configure an xml file that will have the datasource information’s, by default the configuration name hibernate looks for is hibernate.cfg.xml
5. you need to create entity classes which will have tables & columns mappings through annotations
   1. @Entity, @Table, @Column, @Id

Note: Primary key is mandatory

Step1: Creating the maven project

Step2: Adding hibernate & mysql-connector dependencies



pom.xml



hibernate.cfg.xml



Create an employee table in mysql



Creating entity class & mentioning that class in the <mapping> tag of hibernate.cfg.xml

Step 1:

Employee.java



Source Code:

**package** com.cognizant;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

**public** **class** Employee {

@Id

@Column(name = "id")

**private** **int** empId;

**private** String name; // @Column is not required because variable & column names are same

**private** **double** salary; // @Column is not required because variable & column names are same

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

}

Step 2: Mention the entity class in the hibernate.cfg.xml



Step 3:

Performing CRUD operations on the entity.

You will use following instances to perform the CRUD operations

* Configuration cfg = new Configuration();

cfg.configure(); // default chooses hibernate.cfg.xml

cfg.configure(“xml file”);

* SessionFactory factory = cfg.buildSessionFactory();
* Session session = factory.openSession();
* session.save(object1);
* session.save(object2)
* session.get(classname.class, primary\_key)

You will get SSL exception so use below url in the hibernate.cfg.xml



HibernateDemo.java

**package** com.cognizant;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**public** **class** HibernateDemo {

**public** **static** **void** main(String[] args) {

// 1st step is to create Configuration object that loads the xml configuration file

Configuration cfg = **new** Configuration();

cfg.configure();

// 2nd step is to create SessionFactory

SessionFactory factory = cfg.buildSessionFactory();

// 3rd step is to create Session

Session session = factory.openSession();

Employee employee = **new** Employee();

employee.setEmpId(200);

employee.setName("Bruce");

employee.setSalary(34200);

// 4th step is to save the object but you must begin the transaction

Transaction tx = session.beginTransaction();

session.save(employee);

tx.commit();

session.close();

factory.close();

System.***out***.println("DONE!");

}

}

HibernateDemoToRetrieve.java

**package** com.cognizant;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**public** **class** HibernateDemoToRetrieve {

**public** **static** **void** main(String[] args) {

// 1st step is to create Configuration object that loads the xml configuration file

Configuration cfg = **new** Configuration();

cfg.configure();

// 2nd step is to create SessionFactory

SessionFactory factory = cfg.buildSessionFactory();

// 3rd step is to create Session

Session session = factory.openSession();

Employee emp = session.get(Employee.**class**, 100);

System.***out***.println("Id: "+emp.getEmpId()+", Name: "+emp.getName()+", Salary: "+emp.getSalary());

session.close();

factory.close();

System.***out***.println("DONE!");

}

}

Criteria object in hibernate:

It allows you to fetch all the records in a Collection and can also apply some condition while fetching all the records

To create criteria you will use

Criteria crt = session.createCriteria(EntityClass.class);

List list = crt.list();

Exercise:

Create a menu driven program which shows following options

1: Insert

2: Retrieve by Id

3: Display All

4: Delete by Id

5: Update the salary based on Id

6: Exit

Note: Perform all the CRUD operations in a separate class but create this menu in main class

Note: Reuse the Session Factory instance in separate class

Applying the conditions in the criteria

In criteria you can apply lot of conditions like you apply in sql query.

Criteria crt = session.createCriteria(Employee.class)

crt.add(Restrictions.eq(“variableName”, value));

variableName is the property that maps to columns



Output



Till now we didn’t write any queries, but hibernate/ORM supports passing queries also, the queries write in ORM is called as JPQL/HQL

JPQL: Java Persistence Query Language

HQL: Hibernate Query Language, it is same as JPQL.

In JPQL/HQL you write queries for entities not for the tables.

Ex:

from com.cognizant.Employee : means you get all the entities

select emp from com.cognizant.Employee emp: emp is an alias to access entity property

Some of the conditions also you can write with that alias.

Select emp from com.cognizant.Employee emp

where emp.empId = 100

Above query will get employee entity for the employee id 100

The above query can ignore the package names if the classes don’t have any conflicts

i.e.,

select emp from Employee emp where emp.empId = 100;

Here emp refers to the object completely, however you can also access only few properties

Select emp.empId, emp.name from Employee emp;

Now you will get only empId and name but not salary



Output:



You can also get only few properties instead of getting whole entity



Here select e.empId, e.name from Employee e will only have part of the entity, hence it is going to create an object of type Object that will have 2 properties of Object type(id & name), which is why you will typecast to Object[].

Spring Boot:

It allows you to quickly configure the spring application, it gives you an environment where you can simplify the development of the spring application.

Without spring boot, you do following steps

1. You configure xml file that will have bean configurations
2. You configure xml file to perform component scanning
3. You configure the dependencies of the beans in xml file
4. You add required libraries for spring core, spring mvc, spring orm, spring security and etc
5. If it’s a webapplication you
   1. configure the server
   2. configure the front controller
   3. configure the spring configure file and etc

With spring you can avoid all the above steps

It provides you the autoconfiguration feature where the spring beans are automatically configured without xml file

It provides you the embedded server if you want to develop web applications, it means you don’t have to configure the server

It does the configuration based on the library you add in the project

To perform these auto-configurations spring boot provides lot of starter libraries like:-

1. spring boot web starter
2. spring boot jpa starter
3. spring boot actuator starter
4. spring boot security starter and so on

If you add any starter library of spring boot then all the configurations required will be done by the library the only thing you must do is you should use @SpringBootApplication on top of the class

How to create spring boot application

1. Create maven project and manually add the library from maven repository
2. Use Spring Initializr where you will get UI to add the libraries and it also gives you the project structure required for spring boot applications

Below is the project we are creating from spring initializr



Below is the project structure you get once you import this project in eclipse



@SpringBootApplication: this takes care performing autoconfigurations like:

* Component scanning starts from the package where you have @SpringBootApplication, you can have all the components inside com.cognizant or sub-package of it
* It configures DispatcherServlet, Server if you have added spring boot starter web library
* It configures all the dependency injection required for the application depending on the library i.e., Supplying Datasoruce to the HibernateTemplate/JdbcTemplate to perform database activities if you have added spring boot starter jpa

Without Spring Boot  
EmployeeService can be maintained by spring container in two ways

1. <bean> having EmployeeService in the xml file
2. @Service on top the EmployeeService, but you must <component-scan> in the xml file.

How to get the EmployeeService instance

1. @Autowired on top of the EmployeeService field
2. EmployeeService you can also get through getBean() method

Note: getBean() is present inside ApplicationContext, we will create ApplicationContext only if we are using console based program, if we are using web based program no need of creating ApplicationContext

In Console based application

ApplicationContext ctx = new ClassPathXmlApplicationContext(“xmlfile”);

In Web based application

ApplicationContext is created by FrontController, so you can directly auto-wire the beans or get the beans

With Spring Boot

EmployeeService instance is maintained by spring container when it has any of the @Component type annotation like, @Component, @Configuration, @Service, @Repository, @Controller, @RestController

How to get the EmployeeService instance

1. @Autowired on top of the EmployeeService field
2. EmployeeService you can also get through getBean() method

Note: getBean() is a method present in ApplicationContext, however in spring boot you will get the application context object through command line runner which is created using @Bean

EmployeeService.java



Note: This instance will be maintained by the spring container because of component scan and you can get this object through application context from the command line runner when you are using console-based application.

SpringBootDemoApplication.java



Note: ApplicationContext object is supplied by the spring boot application so that you can get all the instances using getBean from the ApplicationContext.

Output:



Note: if you modify the code it wouldn’t be reflected because server will not be having idea about the changes, for that reason you can add another library called Dev tools, that will provide auto-reload options for the server if any changes done in the code

Adding Devtools library



Exercise:

Create one class EmployeeDao and inside that create one test method and autowire the employee dao in the EmployeeService and call test method of EmployeeDao from testEmp method of EmployeeService.

Spring Data JPA

You can simplify performing operations on the database with the predefined interfaces, spring boot will take care of implementing methods for the interfaces.

Spring Data Jpa gives you some interfaces that can perform all the database operations, you can just inherit that interface to get the functionalities, but you don’t have to implement the interface.

Some of the interfaces are:

CrudRepository<T, ID>: it has methods to perform insert, update, delete and retrieve

JpaRepository<T, ID>: it extends CrudRepository + it has methods to perform sorting and pagination.

Note: You don’t have to implement the interface but you just extend the interface the spring boot will take care implementing the interfaces using ORM features.

Adding spring data jpa library and also the mysql library

pom.xml



Once you add spring data jpa library your connection will be automatically established based on the properties you have configured in the application.properties

CrudRepository & JpaRepository has many method to perform crud operation some of them are:

* T save(T t)
* List<T> findAll()
* T findById(ID id)
* void delete(ID id)

Steps:

Firstly me must configure the application.properties to mention datasource information

Secondly we must have entity class mapping to appropriate table

Thirdly we must create an interface which is our own repository that extends either CrudRepository or JpaRepository

Fourthly we can tell spring to inject the object to our repository to perform the operations i.e, in Servie we use @Autowired

Note: Spring Boot has provided some link which has all the possible properties with description you can mention in the application.properties



Modify the application.properties



Create the entity class for the appropriate table

Employee.java



Creating our own repository



Now you can create a reference of EmployeeRepository in the EmployeeService and call any methods of EmployeeRepository

JpaRepository<T, ID> gives you lot of methods to call:

1. List<T> findAll()
2. T save(T)
3. Optional<T> findById(ID)
4. void count()
5. void deleteById(ID)
6. void deleteAll()

EmployeeService.java



Now you can call the fetchAllEmploees from comandLineRuner



How to store the data in spring data jpa

save(object) returns object that is saved

How to retrieve the data by id

findById(ID) returns Optional so that you can get the entity or null

How to delete the data by id

delete(ID) returns void

Apart from this you can also write custom queries in the repository on top of the methods so that if you call that method the query written on top would be executed

Ex:

@Query(“select e from Employee e where e.name = ?1”)

List<Employee> getEmployeesByName(String name);

Now you don’t have to implement the getEmployeesByName instead you just call the method you will get the List<Employee> based on the name you pass

Storing the employee instance

EmployeeService.java



Main class



Assignment is:

1: insert

2: fetch employee by id

3: fetch all employees

4: delete by id

5: update salary based on id

6: exit

Loop until you enter 6

How to create our own methods and perform operations:



Now you can call these methods from EmployeeService



Call from commandLineRunner



List of class-level, method-level, field level annotations

|  |  |  |
| --- | --- | --- |
| Class level | Method level | Field level |
| @Component | @Bean | @Value |
| @Service | @RequestMapping | @NotNull |
| @Controller | @Autowired | @Autowired |
| @RestController | @GetMapping | @Qualifier |
| @Repository | @PostMapping | @OneToOne |
| @Configuration | @PutMapping | @OneToMany |
| @SpringBootApplication | @DeleteMapping | @ManyTOMany |
| @Import | @OneToOne |  |
| @ImportResource | @OneToMany |  |
| @EnableAutoConfiguration | @ManyTOMany |  |
| @ComponentScan |  |  |
| @RequestMapping |  |  |
| @Transactional |  |  |

Performing @OneTo@One, @OneToMany, @ManyToMany, @ManyToOne annotations

These annotations are helpful in joining the entities reduces writing complex join queries these annotations are part of Jpa which can be used in any ORM framework as well in spring data jpa also.

@OneToOne: one entity mapped to one entity

Ex: One Employee has One Address

@OneToMany: one entity mapped to multiple entities

EX: One State has many Cities

@ManyToOne: Many entities mapping to One entity

EX: Many Cities belonging to same state

@ManyTOMany: Many entities mapping to many entities, however it is a combination of one to many in both the tables

EX: many students taking many courses, in that one student enrolled to many courses and same course could be enrolled by many students.

One to One extra



Below are the employees



Below are the address



Create a new project



application.properties

server.port=8081

spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/cts\_2021?useSSL=false

spring.datasource.username = root

spring.datasource.password = root

Create the command line runner in Main class



Address entity

**package** com.cognizant.model.entities;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToOne;

**import** javax.persistence.Table;

@Entity

@Table(name = "address")

**public** **class** Address {

@Id

@Column(name = "address\_id")

**private** **int** addressId;

**private** String state;

**private** String city;

@JoinColumn(name = "emp\_id")

@OneToOne

**private** Employee empIdLink;

**public** **int** getAddressId() {

**return** addressId;

}

**public** **void** setAddressId(**int** addressId) {

**this**.addressId = addressId;

}

**public** String getState() {

**return** state;

}

**public** **void** setState(String state) {

**this**.state = state;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String city) {

**this**.city = city;

}

@Override

**public** String toString() {

**return** "Address [addressId=" + addressId + ", state=" + state + ", city=" + city + "]";

}

**public** Employee getEmpIdLink() {

**return** empIdLink;

}

**public** **void** setEmpIdLink(Employee empIdLink) {

**this**.empIdLink = empIdLink;

}

}

Employee.java

**package** com.cognizant.model.entities;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.FetchType;

**import** javax.persistence.Id;

**import** javax.persistence.OneToOne;

@Entity

**public** **class** Employee {

@Id

@Column(name = "id")

**private** **int** empId;

**private** String name;

**private** **double** salary;

@OneToOne(cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***, mappedBy = "empIdLink")

**private** Address address;

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

@Override

**public** String toString() {

**return** "Employee [empId=" + empId + ", name=" + name + ", salary=" + salary + ", address=" + address + "]";

}

**public** Address getAddress() {

**return** address;

}

**public** **void** setAddress(Address address) {

**this**.address = address;

}

}

Create the repository to perform crud operations on employee/address

Note: You can create AddressRepository to perform CRUD on address entity however you don’t have to create this as you are not independently working on address & you can create EmploeeRepository to perform CRUD operation employee entity.

EmployeeRepository.java

**package** com.cognizant.model.dao;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.cognizant.model.entities.Employee;

**public** **interface** EmployeeRepository **extends** JpaRepository<Employee, Integer>{

}

EmployeeService.java

**package** com.cognizant.model.service;

**import** java.util.List;

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

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

**import** com.cognizant.model.dao.EmployeeRepository;

**import** com.cognizant.model.entities.Employee;

@Service

**public** **class** EmployeeService {

@Autowired

**private** EmployeeRepository employeeDao;

**public** Employee fetchEmployee(**int** id) {

**return** employeeDao.findById(id).get();

}

**public** List<Employee> fetchEmployees() {

**return** employeeDao.findAll();

}

}

Call these methods in commandLineRunner

**package** com.cognizant;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.Bean;

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.service.EmployeeService;

@SpringBootApplication

**public** **class** SpringDataJpaMappingsApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringDataJpaMappingsApplication.**class**, args);

}

@Bean

**public** CommandLineRunner runner(ApplicationContext context) {

**return** (args) -> {

EmployeeService service = context.getBean(EmployeeService.**class**);

System.***out***.println("------------------ Getting a particular employee --------------------");

Employee e = service.fetchEmployee(700);

System.***out***.println(e);

};

}

}

Exercise:

Try out other types of mapping with different tables

1. One to Many with States & Cities table
2. Many to Many with Course & Students table

Hint: you must have List reference to hold many entities

Implementing One to Many

States table



Cities table



Cities.java

**package** com.cognizant.model.entities;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

**public** **class** Cities {

@Id

@Column(name = "city\_id")

**private** **int** cityId;

@Column(name = "city\_name")

**private** String cityName;

**public** **int** getCityId() {

**return** cityId;

}

**public** **void** setCityId(**int** cityId) {

**this**.cityId = cityId;

}

**public** String getCityName() {

**return** cityName;

}

**public** **void** setCityName(String cityName) {

**this**.cityName = cityName;

}

@Override

**public** String toString() {

**return** "Cities [cityId=" + cityId + ", cityName=" + cityName + "]";

}

}

States.java

**package** com.cognizant.model.entities;

**import** java.util.List;

**import** javax.persistence.CascadeType;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.FetchType;

**import** javax.persistence.Id;

**import** javax.persistence.JoinColumn;

**import** javax.persistence.OneToMany;

@Entity

**public** **class** States {

@Id

@Column(name = "state\_id")

**private** **int** stateId;

@Column(name = "state\_name")

**private** String stateName;

@OneToMany(cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***)

@JoinColumn(name = "state\_id\_ref")

**private** List<Cities> cities;

**public** **int** getStateId() {

**return** stateId;

}

**public** **void** setStateId(**int** stateId) {

**this**.stateId = stateId;

}

**public** String getStateName() {

**return** stateName;

}

**public** **void** setStateName(String stateName) {

**this**.stateName = stateName;

}

**public** List<Cities> getCities() {

**return** cities;

}

**public** **void** setCities(List<Cities> cities) {

**this**.cities = cities;

}

@Override

**public** String toString() {

**return** "States [stateId=" + stateId + ", stateName=" + stateName + ", cities=" + cities + "]";

}

}

StatesRepository.java

**package** com.cognizant.model.dao;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.cognizant.model.entities.States;

**public** **interface** StatesRepository **extends** JpaRepository<States, Integer> {

}

StatesService.java

**package** com.cognizant.model.service;

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

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

**import** com.cognizant.model.dao.StatesRepository;

**import** com.cognizant.model.entities.States;

@Service

**public** **class** StatesService {

@Autowired

**private** StatesRepository statesDao;

**public** States fetchState(**int** stateId) {

**return** statesDao.findById(stateId).get();

}

}

CommandLineRunner code

**package** com.cognizant;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.Bean;

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.entities.States;

**import** com.cognizant.model.service.CustomerService;

**import** com.cognizant.model.service.EmployeeService;

**import** com.cognizant.model.service.StatesService;

@SpringBootApplication

**public** **class** SpringDataJpaMappingsApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringDataJpaMappingsApplication.**class**, args);

}

@Bean

**public** CommandLineRunner runner(ApplicationContext context) {

**return** (args) -> {

// EmployeeService service = context.getBean(EmployeeService.class);

// System.out.println("------------------ Getting a particular employee --------------------");

// Employee e = service.fetchEmployee(200);

// System.out.println(e);

// System.out.println("------------------Getting all the customers--------------------------");

// CustomerService customerService = context.getBean(CustomerService.class);

// customerService.fetchCustomers().forEach(item -> System.out.println(item));

StatesService statesService = context.getBean(StatesService.**class**);

States states = statesService.fetchState(2);

System.***out***.println(states);

};

}

}

Lombok:

It provides you a feature where you can reduce your code to be more concise by not writing certain boiler plate code, i.e,. you can avoid writing setters, getters, constructors, toStirng, equals, hashCode, validations like null check and so on, Lombok provides some annotations using which you can generate these codes without writing it.

Lombok provides some annotations like:

@Data: setters & getters, you can write on top of the class

@Setter & @Getter: It is also for setters & getters but this can be written on top the fields

@ToString:

@NoArgsConstructor, AllArgsConstructor

Official website of Lombok

<https://projectlombok.org/>

pom.xml



Users.java



Now you will get setters & getters for all the fields



We could able to call setters & getters without writing because of Lombok.

If Lombok doesn’t generate setters & getters you can use java -jar <file-name>.jar as below:



Now a wizard will open you need to locate the eclipse installation folder



After you click on install/update you will get complete wizard



Note: At the end update the maven project and check the Lombok features

Using AllArgsConstructor to generate the parameterized constructor



Lombok provides you the feature of null checking, so that you can avoid writing code to handle null values

@NonNull can be used on top of the variable as well in the parameter also.



If you pass null to the name then you will get NullPointerException.

Exercise:

Try to use Lombok features in the spring data jpa entities

Angular Framework

It is used to develop front end applications like mobile, web, native, desktop using 2 main languages

1. Typescript
2. HTML

Using angular you can develop Single Page Applications.

Single Page Applications

These are the applications where everything happens in one single page, you can develop single page applications with the help of components.

Components: These are the things what you see in the page and they are reusable it means you can add a component to another component, they are easily maintainable.

Software’s required

1. Node.js:

<https://nodejs.org/en/>

1. Visual Studio Code:

<https://code.visualstudio.com/download>

1. Angular-CLI (Tool used to quickly create & develop angular application)

How to install angular/cli

>> npm install -g @angular/cli

>> npm install @angular/cli

Two commands you get when you install node.js

1. node: It is used to run javascript code
2. npm: It is used download javascript libraries/tools from the internet ex: angular/cli, create-react-app, bootstrap, jquery.

Verifying the @angular/cli installation

ng version



How to create angular projects

To create the angular projects, you use

*ng new project-name*

Here,

ng: is angular command

new: is to create new project

project-name: is the name of the project like my-first-app, online-shopping-ui, app-demo and so on.

What happens when you enter ng new project-name

Once you enter this command angular/cli downloads all the necessary libraries/tools to develop angular applications.

1. node\_module folder: set of javascript libraries to develop angular application
2. lite server: an embedded server, used to run the angular application on this server
3. typescript compiler: to compile the typescripts to javascripts
4. auto-compilation and live reload features: it is like a development tool which compiles when you code and reloads the application
5. Project structure with best practices
6. Many other things like package.json, angular.json, main.ts, index.html, polyfills.json.

To understand and develop angular applications we need certain pre-requisites like

1. HTML
2. CSS
3. Javascript new version ES6 features
4. Typescript

Understanding ES6

ES stands for ECMAScript, which is a specification/standard to the Javascript, the new feature of ES6 simplifies writing javascript code by providing some new generation syntax in javascript.

Below are the ES6 features:

1. Keywords like let & const to declare variables
2. Keywords like class, super, constructor, extends to code in an object-oriented fashion
3. Template strings
4. Arrow functions
5. Rest & Spread operators

Note: ES is understood by both browser & node.js as both are runtime environment for Javascript.

How do you run javascript in browser?

* One is through developer tools in the browser
* Another way is including javascript in html and open html page in browser.

How do you run javascript in node.js?

* You can create javascript file and run that file using node command
* But in node.js you will not see browser you will see only the console.



Keywords like let & const

Earlier javascript was allowing to declare variables using var, which doesn’t have any scope, hence ES6 introduced two keywords let & const to create block scoped variables, which are visible only within the block

let variables can be modified however const can’t be modified



You can’t access y and z outside the loop, because they are visible only within the for loop.

Keywords like class, constructor, super, extends

Earlier you can create object using a function and add function to the object using prototype as below:



In ES6 you can use class, constructor keywords to create a class and write functions inside the class, so that the objects of the class can call these functions.



Inheritance in javascript



Template Strings:

It allows you to create the strings and concatenate the value without breaking the string.



Arrow Function (=>): It simplifies writing anonymous functions / callback functions in javascript.

Callback functions performs some actions in a function like connecting to server or performing some action based on some events or manipulating the data and etc.

Earlier you were writing the callbacks using function() keyword followed by { } braces, but in new edition you can avoid writing function keywords and {} braces if the functions needs only one statement implement.

Old approach & arrow function of using forEach of an array in javascript



Use arrow function in map() function and generate a new array from the existing array.

existingArray = [1, 2, 3, 4, 5];

newArray = [10, 20, 30, 40, 50]

Solution:



Rest & Spread operators

Rest allows a variable to accept 0 or more inputs

RestDemo



In the above example …y accepts 0 or more values

Spread operator

Spread operator accepts single input variable to be distributed to multiple parameters, but the single input variable must be an array



Here …items have an array 5 elements that are spread across 5 parameters however if you have rest operator then you may not need 5 parameters, because rest operator can accept the rest of the inputs, in the above example: items values 1 & 2 are accepted by a & b, however 3, 4, 5 are accepted by single rest variable c.

Typescript:

It is a super set of javascript, it adds types to the javascript so that data’s are more reliable and you can control the data and function to have specific type of values, you need to create typescript in ts file however browsers & node.js doesn’t understand typescripts hence it has to be converted to the javascript.

Note: You will not work in javascript instead you will work in the typescript and compile it to get the javascript & consider javascript as an executable file generated from the typescript.

Some of the features of typescript

1. Typescript gives types to the variables & functions parameters, return types like
   1. number
   2. string
   3. boolean
   4. undefined
   5. any
   6. Array
2. Typescripts handle errors at the compilation time itself
3. Typescripts supports all the features of javascript like ES6 features, functions, loops, conditions etc, but javascript doesn’t support all the typescript features
4. Apart from types typescript supports other features like
   1. Short cut initializations in a constructor
   2. Exports & imports of classes, functions, variables
   3. Interfaces, Decorators that adds extra benefits while coding

Manually installing typescript compiler

>> npm install -g typescript

Verifying the typescript

>> tsc --version

Note: If you are using @angular/cli then you don’t need to install typescript compiler.

hello.ts





arrowFunctionDemo.ts



After compilation you can see the generated js code is having ES5 syntax, this is the default behaviour of typescript compiler, but you can generate ES6 syntax code while compiling.

arrowFunctionDemo.js



How to compile TS to generate ES6 syntax of javascript

>> tsc --target es6 file.ts

The above command generates the javascript which is compatible to the ES6 feature.

Writing the classes, constructors & functions in typescript



Above code has a variable created inside the class & also initializing them inside constructor, however in typescript this can be still reduced with the help of short-cut initialization constructor



Note: Inside constructor parameter you can either use private or public

You can use export for a class which needs to imported in another class

exportDemo.ts



importDemo.ts



Angular Framework

It is used to develop single page applications using HTML & Typescript, it allows you to develop different types applications like web, mobile, desktop, native.

Single page applications:

It is an application where everything happens in one page and when any changes is done it reloads only the part of the page instead of reloading the entire page.

Ex: Gmail, Facebook, Github, Angular.io and so on.

Angular uses component-based approach, where components are the user interfaces which can be independently developed and can be reused in another component, it would be easy to change when its independent from others

A single application is a collection of such components together, it will have a root component inside that many other components will be added.

Ex: Facebook profile is a component that is reused in comment, post, like and user profiles

Ex: Ratings in amazon

Angular team has provided Angular/CLI tool kit

It provides following benefits

1. It allows you to quickly create & develop angular application
2. It creates a ready to run/deploy/test application
3. It provides you an embedded development server to host the application
4. It provides you the typescript compiler internally in the project
5. It downloads all the required libraries to develop angular application
6. It provides you auto-compilation feature
7. It provides you auto-reload feature
8. It provides commands that can easily create angular build blocks required for your angular application in one single command

To install angular/cli you need node.js

The command you must use to install is

>> npm install -g @angular/cli

The command to verify the angular installation is

>> ng version

How to create a new angular project

>> ng new app-name

How to run the angular project

>> ng serve (you must enter this command from the root directory of the project, ie., app-name is the root directory).



Open the project folder ‘my-app’ from the VS Code



In every new angular project you will have one default root component with some welcome message, the only page that will shown is index.html and it will have the root component (<app-root>).

Components: It will have one tagname and content for the tag name

How to create a component

You will use @Component({}) decorator, it should be written on top of a class, The @Component({}) should have minimum two properties

* selector: Provides the tag name
* templateUrl: Provides the content for the selector, i.e., html file location

For tomorrow’s agenda:

1. Creating our own components
2. Understanding different angular building blocks
3. Understanding the commands of angular
4. Understanding of inline template
5. Understanding of data binding.

Creating our own components

You can create components in 2 ways

1. You can manually create, but here you must declare components in the app.module.ts manually and also you must create all the necessary files manually like css, html, ts
2. You can use the command to create, if you use this the command will automatically add the components to the module and also creates all the necessary files like css, html, ts

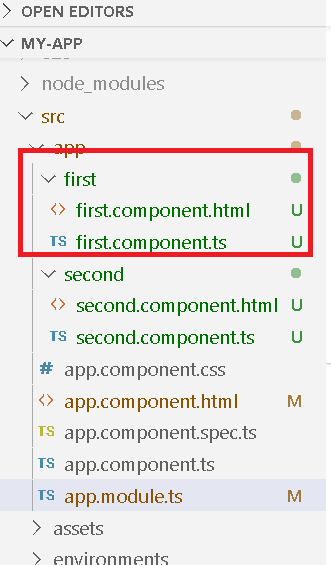
Root Module(app.module.ts):

By default, every angular project will have a root module that represents your entire angular application, it will have entries of all the components, pipes, routers, services, other sub-modules and so on.

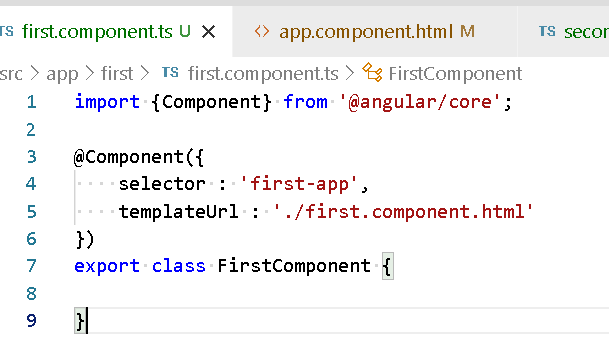
Root Module is bootstrapped at the time loading the application.

Note: Modules can also be created using @NgModule, however you need to mention that in the Root Module

Creating components manually.



first.component.ts

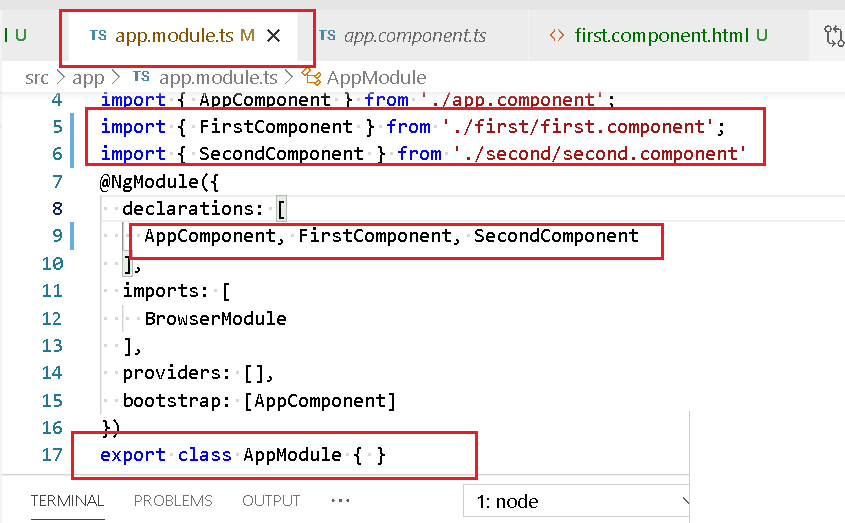


first.component.html



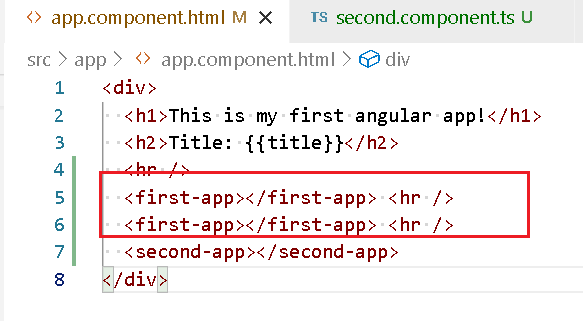
Mentioning the first component in the root module

app.module.ts



Reusing the first-app in another component

app.component.html



Note: second component must also be created

Exercise:

Create a component with 2 variables name and age and display those name & age

How to generate the components through commands

>> ng generate component componentName

(or)

>> ng g c componentName

What happens when you enter this command & from where you should enter this command.

* You need enter this command from the project directory
* When you enter this command angular framework creates components, html, css, spec.ts files as well it declares the component in the app.module.ts.

Angular application structure & the flow

*ng serve* -> angular.json -> finds out the entry point file (main.ts & index.html)

index.html is loaded and main.ts code is executed.

main.ts has the code to bootstrap the AppModule, it means the entire application will launched.

AppModule contains all the elements of the application like components, browser-module, root component to bootstrap and so on.

Some of the important files & folders in angular project

package.json: It will have all the dependencies required for your project

angular.json: It will have all the angular configurations like entry point file, output folder name, global styles configuration and etc.

node\_modules: all the libraries required for the projects, like @angular/core, @angular/common/http, @angular/forms, @angular/compiler and so on.

index.html: This is the only file that will be served to the end user, it will have only the root components

main.ts: It will have the code to launch the application

tsconfig.json: It will have the typescript configurations required for your angular project.

Angular building blocks:

These are the main elements of angular to create the application

1. Components
2. Modules
3. Pipes
4. Databinding
5. Forms
6. Directives
7. Routers
8. Services

Pipes (|): It is used to format the output, angular has many inbuilt pipes like

1. currency
2. date
3. uppercase
4. lowercase
5. json

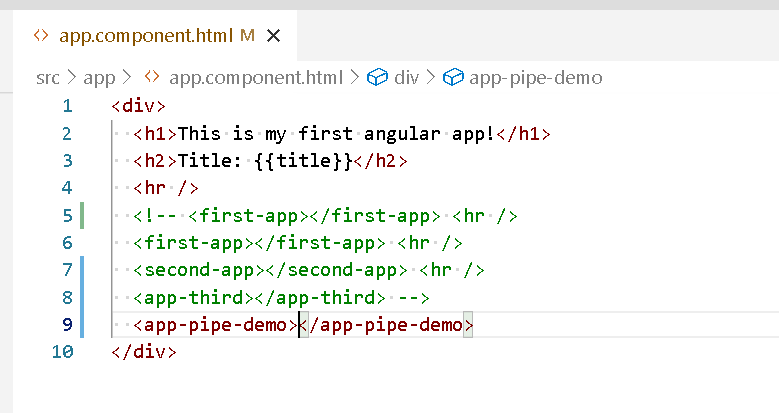
{{data | pipeName}}

Example on pipe

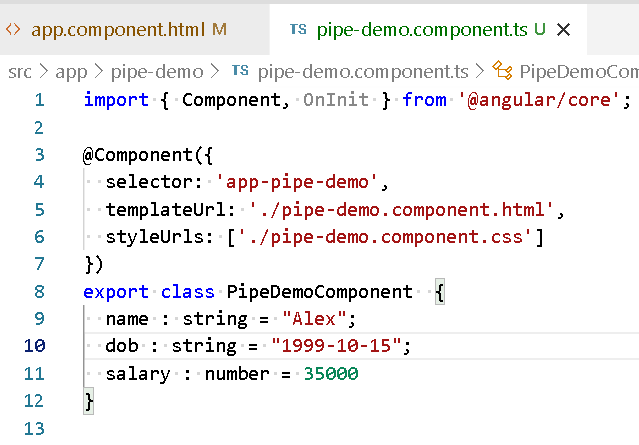
>> ng g c pipeDemo

Mention the pipeDemo component in the root component

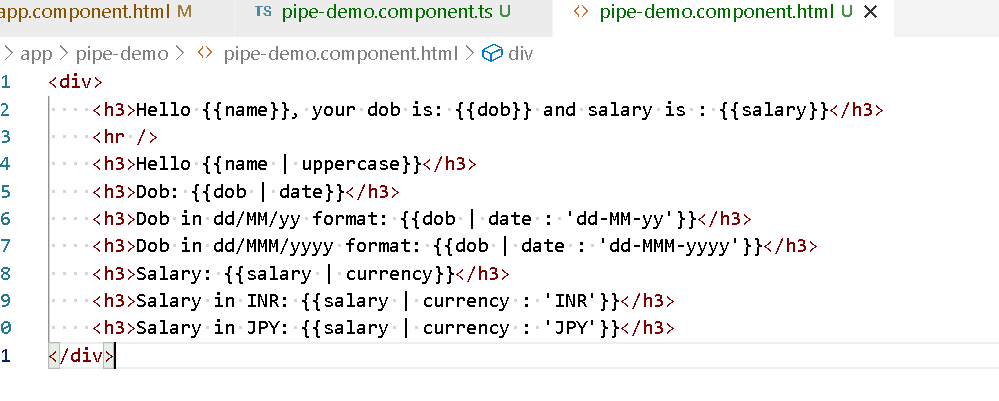
app.component.html



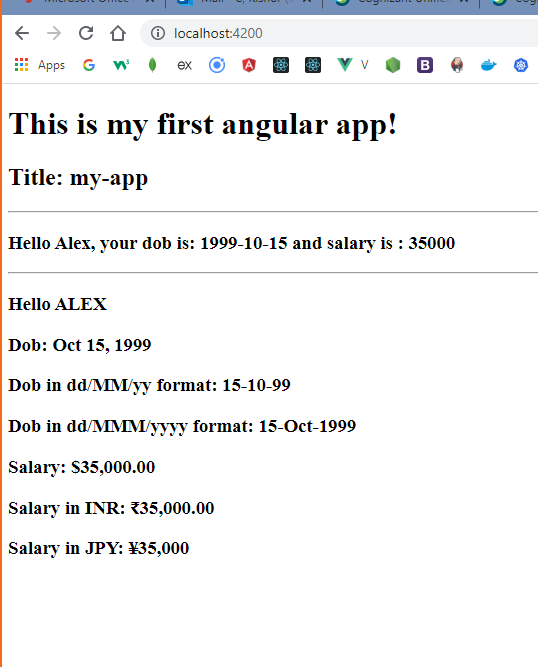
pipe-demo.component.ts



pipe-demo.comoponent.html



Output:



Directives:

Angular has structural directives to work on the DOM elements, the structural directives begins with ‘\*’, it is used as html attributes, some of them are:

1. \*ngFor
2. \*ngIf

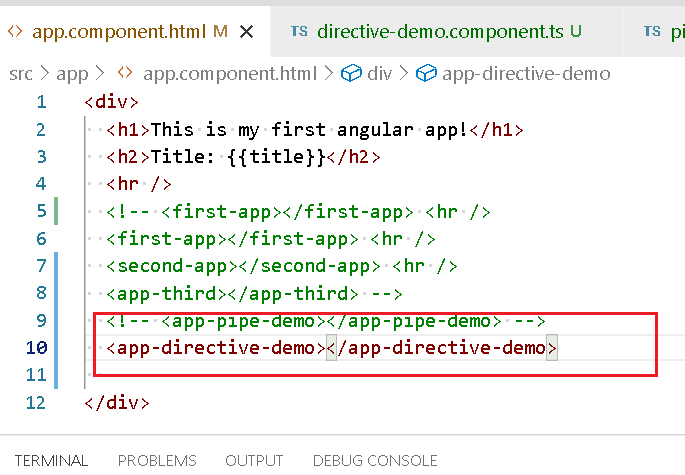
Both of these can be used in HTML

\*ngFor: it is to iterate the items in the array

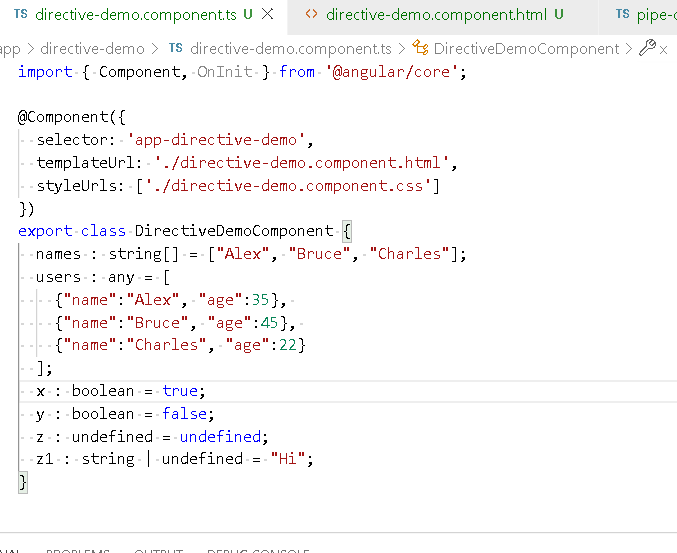
\*ngIf: it is to apply some conditions on the html elements

>> ng g c direcrtiveDemo

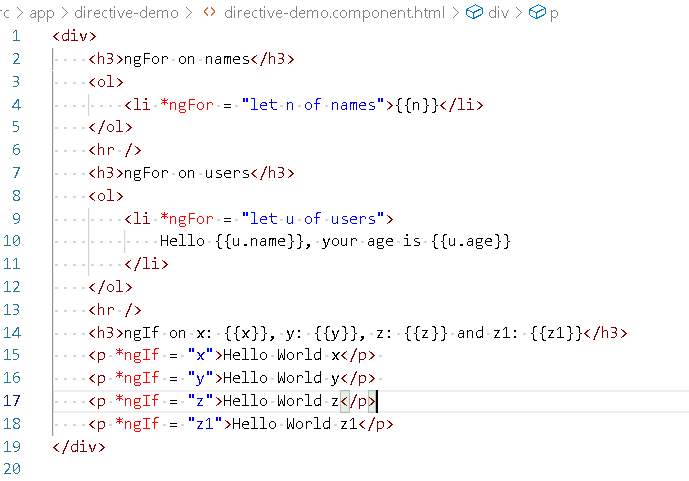
app.component.html



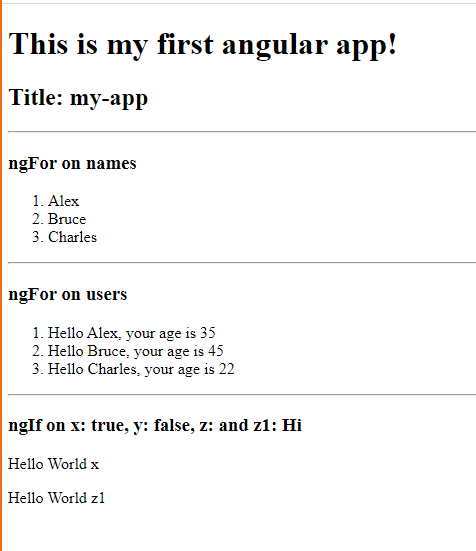
directive-demo.component.ts



directive-demo.component.html



Output:

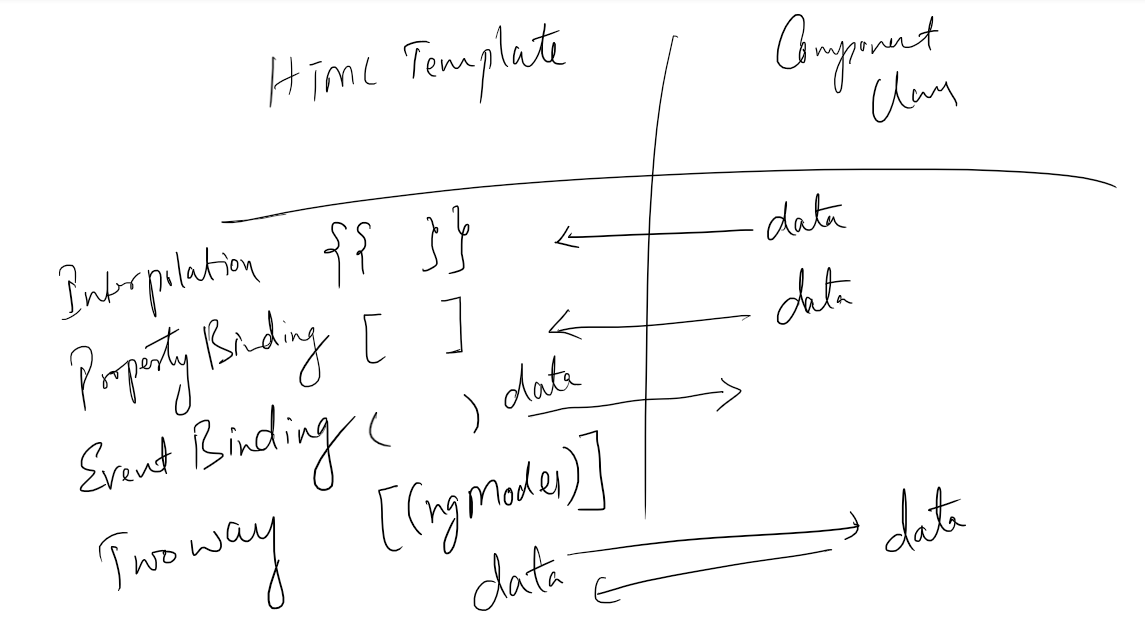


Data binding:

It helps in binding the data between view template(HTML) and the component class, following are the data binding types:

1. Interpolation: {{}}
2. Property binding: []
3. Event binding: ()
4. Two-way data binding: [(ngModel)].

The purpose of the data binding is to share the data between the view template & the component class either in one-direction or both the direction.



Databinding is of two types

1. One-way data binding
   1. Interpolation
   2. Property binding
   3. Event binding
2. Two-way data binding
   1. [(ngModel)]

Interpolation: It is used to bind the data from component to view template, it is mainly used to display the data in the view template

Ex: {{propertyName}} or {{variableName}}

Event binding: It is used to bind the data from view template to the component, it is mainly used to change the data of the component or pass the data to the component from view template

Ex: <button (click) = “fun()”>Click Here</button>

Ex: <button (click) = “fun(values)”>Click Here</button>

Ex: <form (ngSubmit) = “fun(values)”>..</form>

Ex: <input (input) = “fun(values)”>

Ex: <select (change) = “fun(values)”>……</select>

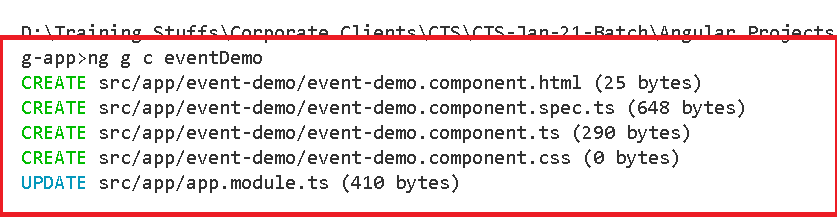
Property binding: It is used to bind the data from component to view template, it is mainly used to manipulate the DOM properties like hidden, disabled, innerText and so on

Ex: <button [disabled] = “propertyName”>Click</button>

Ex: <p [hidden] = “propertyName”>some content</p>

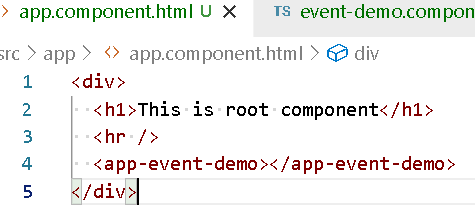
Note: hidden & \*ngIf both will do the same task, however there’s a slight difference you can observe in the DOM structure

Event binding example:

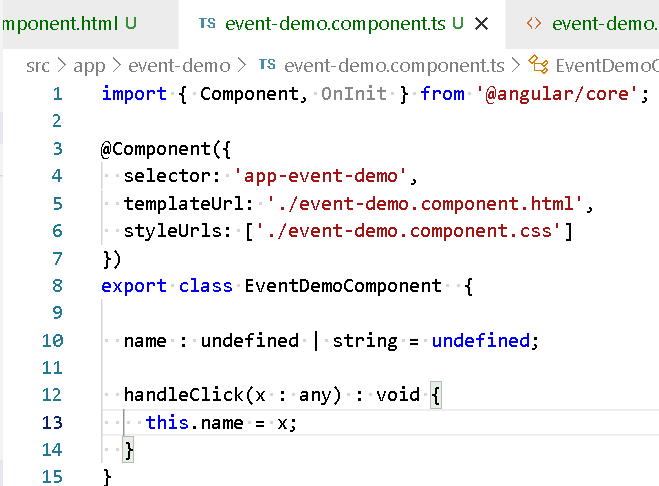


Add the event binding demo into the root component

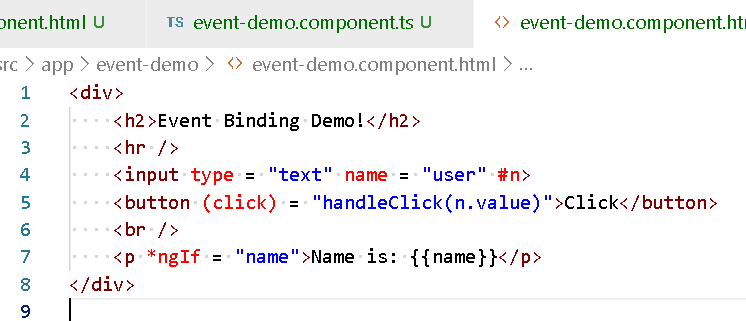
app.component.html



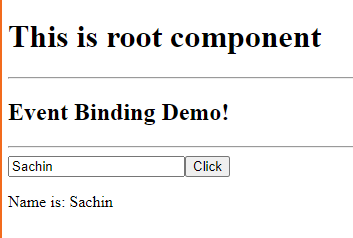
event-demo.component.ts

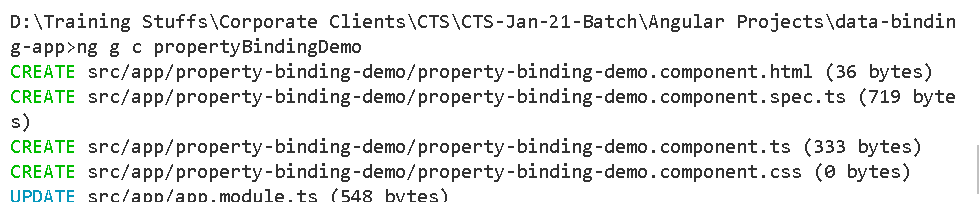


event-demo.component.html

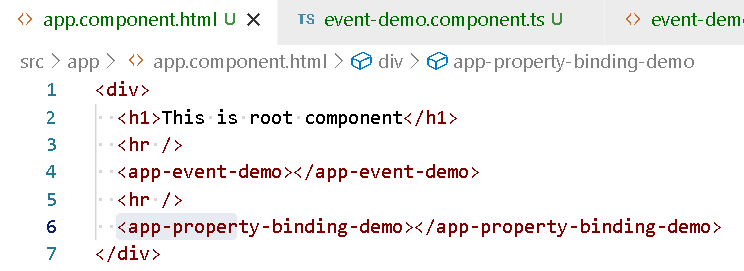


Output:

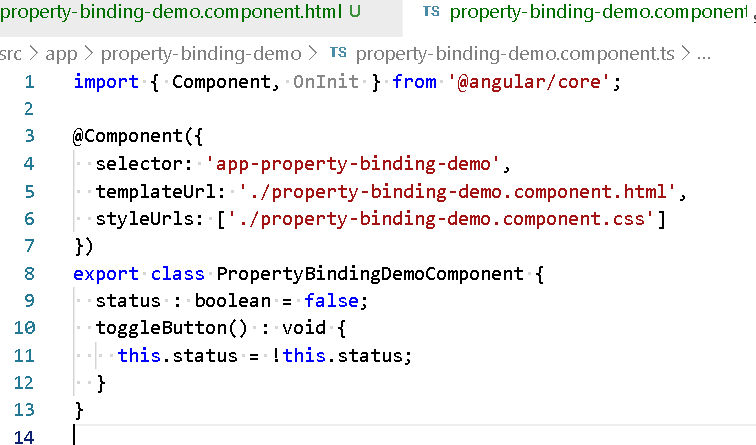


Property Binding Example  


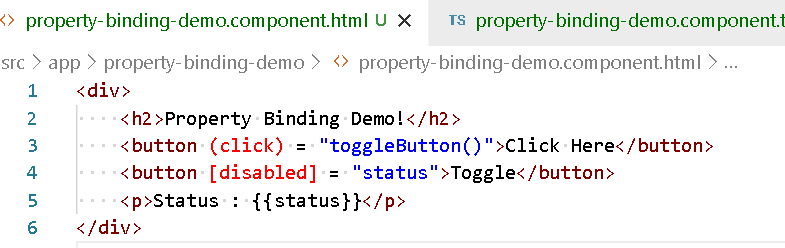
Add property binding demo component in the root component



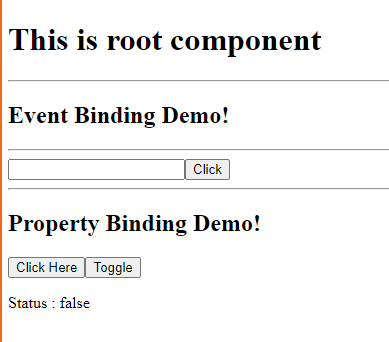
property-binding-demo.component.ts



property-binding-demo.html



Output:

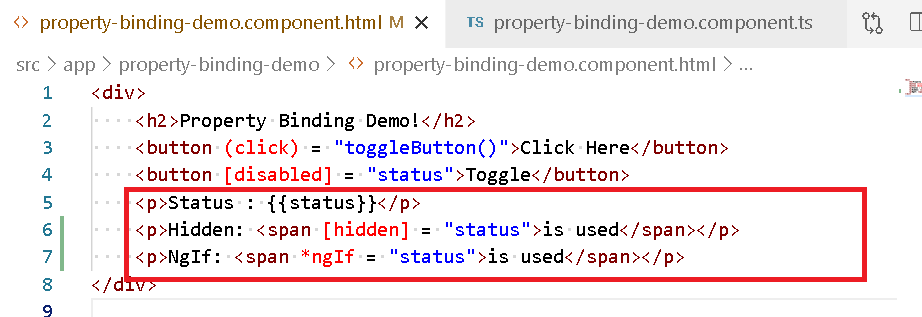


Few other properties you can bind are:

[hidden]: it is to hide the elements

[innerText]: it is to update the elements content

Using \*ngIf & hidden both in property binding component



\*ngIf will render the element on the DOM and to the user interface when it is true, but removes the element from the DOM and user can’t see that element when it is false

Hidden will be always loaded on the DOM however user may see the content if it is false, but will not see the content if it is true, however the DOM will always have that element and only add/removes hidden property to the element

Two-way data binding:

It is used to share the data in both the directions from view template to the component and vice versa.

You must use [(ngModel)] directive to achieve two way databinding, here by default angular will not recognize the [(ngModel)], because it is part of the FormsModule, you must add this FormsModule in the imports of @NgModule.

In the app.module.ts

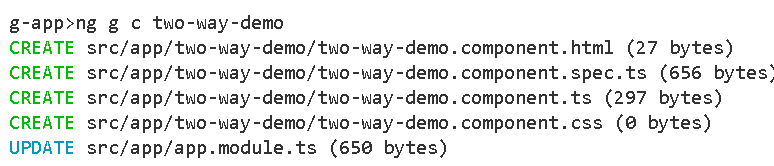
You must add FormsModule in the imports of @NgModule({})

@NgModule({  
…

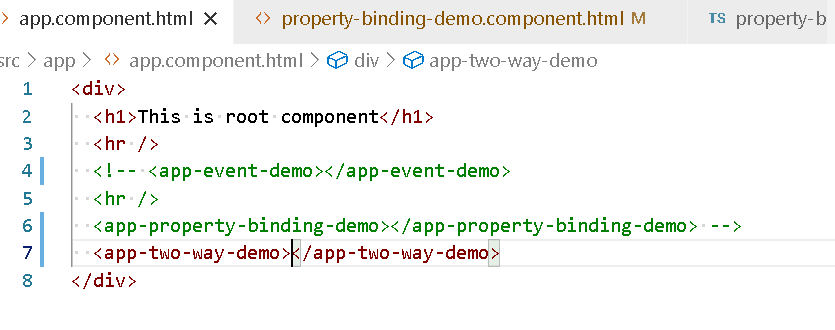
imports : […, FormsModule,…]

…  
  
})

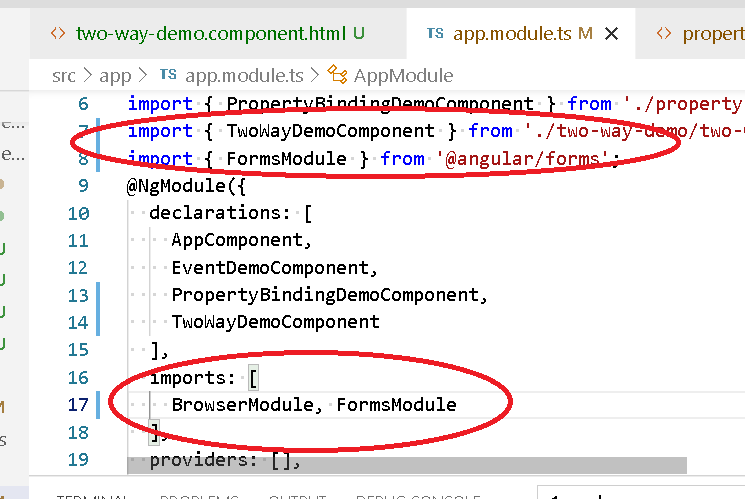
Two-way data binding example:



Adding the two way demo component into the root module



Adding FormsModule in app.module.ts

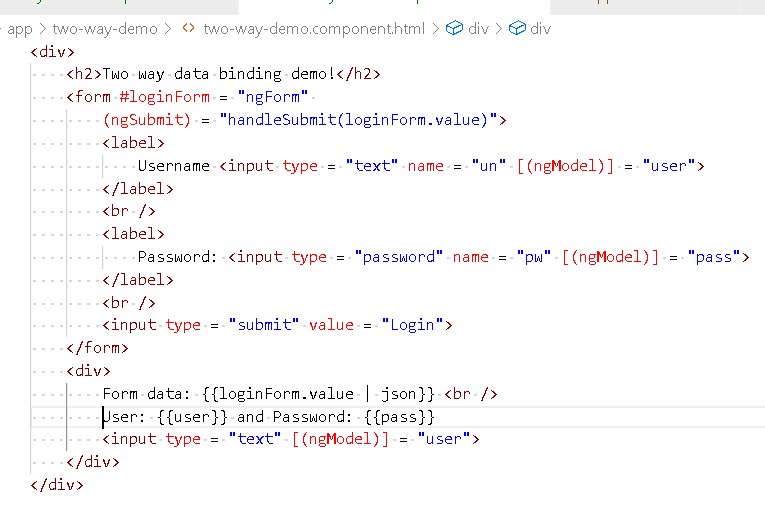


FormsModule helps in recognizing ngForm, ngModel, ngSubmit and etc.

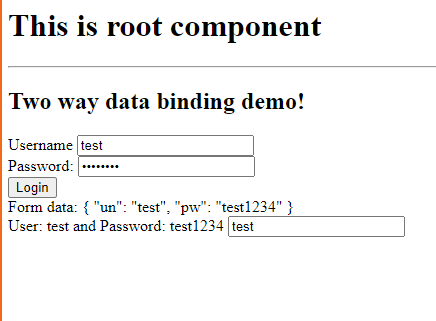
two-way-demo.component.ts



two-way-demo.component.html



Output:

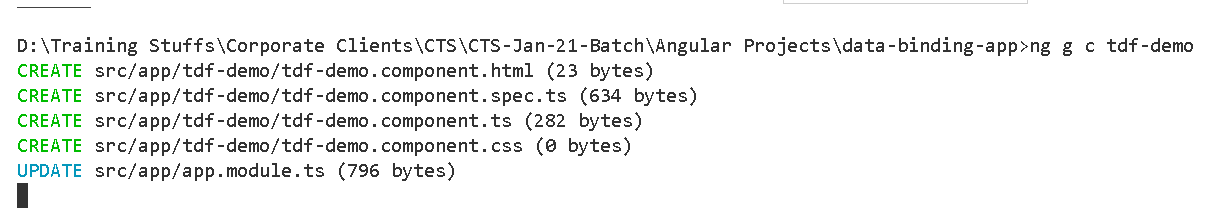


Angular Forms:

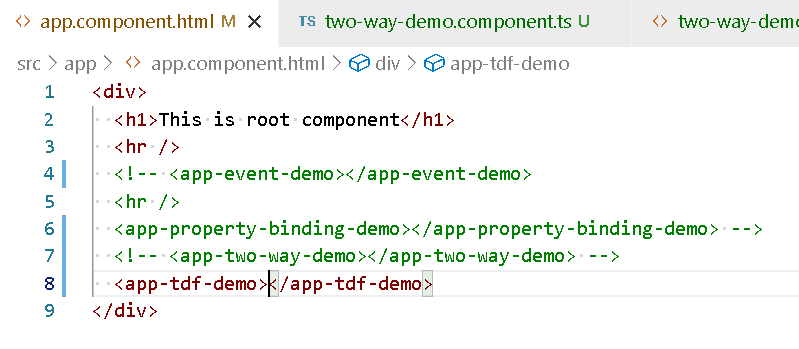
There are two types of forms

1. Template Driven Form (TDF): Here you handle form data, validations in the HTML
2. Model Driven Form (MDF) or Reactive Form: Here you handle form data, validations in the typescript

Template Driven Form example



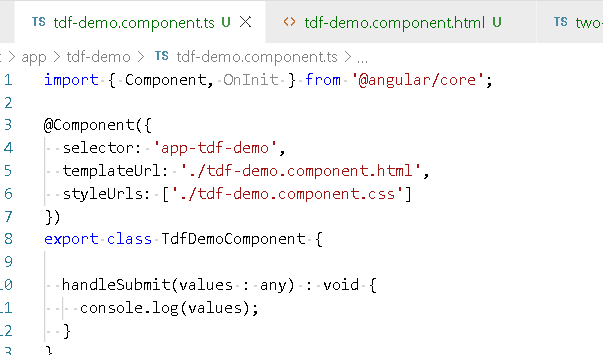
Add the tdf-demo component to the app.component.html



Performing Form Validation in the template driven form using ngModel with valid, invalid, pristine properties

Pristine: it checks the control not modified status, it is true if not modified it will be false if modified.

tdf.demo.component.ts



tdf.demo.component.html

<div>

    <h2>Template Driven Form</h2>

    <div>

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

            <label>

                Enter Name: <input type = "text" name = "user" autocomplete="off" #tf1="ngModel" ngModel required >

                <span style = "color:red" [hidden] = "tf1.valid || tf1.pristine">Username is mandatory</span>

            </label>

            <br />

            <label>

                Enter Password: <input type = "password" name = "pass" #tf2="ngModel" ngModel  required >

                <span style = "color:red" [hidden] = "tf2.valid || tf2.pristine">Password is mandatory</span>

            </label>

            <br />

            <input type = "submit" value = "Login" [disabled] = "userForm.invalid">

        </form>

        <div>

            <h3>Template Reference 1</h3>

            <p>Value: {{tf1.value}}, Valid: {{tf1.valid}}, Invalid: {{tf1.invalid}}, Pristine: {{tf1.pristine}}</p>

            <hr />

            <h3>Template Reference 2</h3>

            <p>Value: {{tf2.value}}, Valid: {{tf2.valid}}, Invalid: {{tf2.invalid}}, Pristine: {{tf2.pristine}}</p>

            <h3>Form Reference</h3>

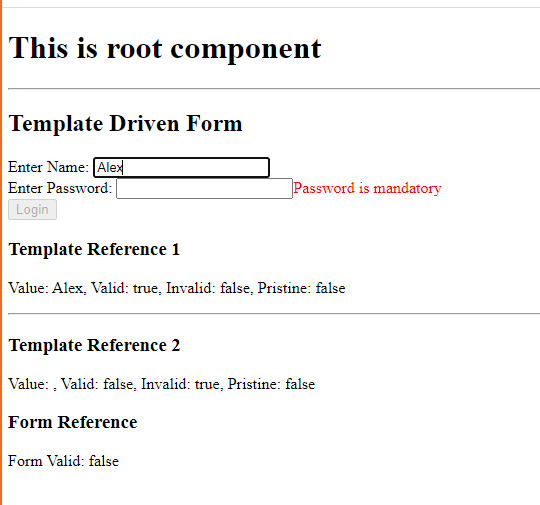
            <p>Form Valid: {{userForm.valid}}</p>

        </div>

    </div>

</div>

Output:

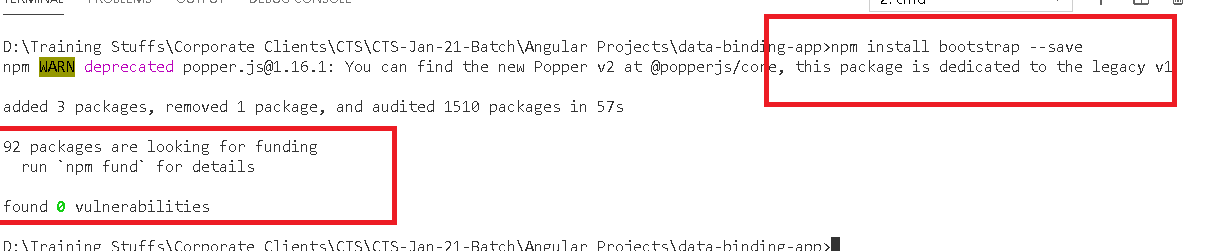


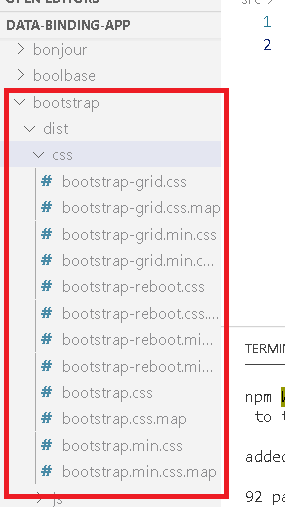
How to install bootstrap.js to your project

You need to use >> npm install bootstrap --save

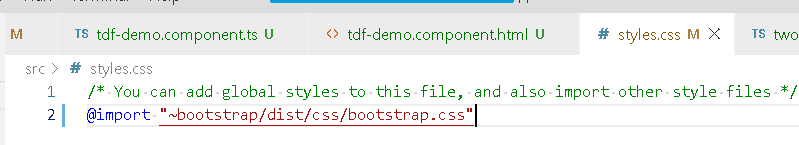
You need to import in the styles.css

Installing bootstrap





Importing the bootstrap.css in the styles.css



Using some of the bootstrap class in the existing template driven form

tdf-demo.component.html

<div>

    <h2>Template Driven Form</h2>

    <div class = "form-group">

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

            <label>

                Enter Name: <input class = "form-control" type = "text" name = "user" autocomplete="off" #tf1="ngModel" ngModel required >

                <span class = "alert alert-danger" [hidden] = "tf1.valid || tf1.pristine">Username is mandatory</span>

            </label>

            <br />

            <label>

                Enter Password: <input class = "form-control" type = "password" name = "pass" #tf2="ngModel" ngModel  required >

                <span class = "alert alert-danger" [hidden] = "tf2.valid || tf2.pristine">Password is mandatory</span>

            </label>

            <br />

            <input class = "btn btn-primary" type = "submit" value = "Login" [disabled] = "userForm.invalid">

        </form>

        <div>

            <h3>Template Reference 1</h3>

            <p>Value: {{tf1.value}}, Valid: {{tf1.valid}}, Invalid: {{tf1.invalid}}, Pristine: {{tf1.pristine}}</p>

            <hr />

            <h3>Template Reference 2</h3>

            <p>Value: {{tf2.value}}, Valid: {{tf2.valid}}, Invalid: {{tf2.invalid}}, Pristine: {{tf2.pristine}}</p>

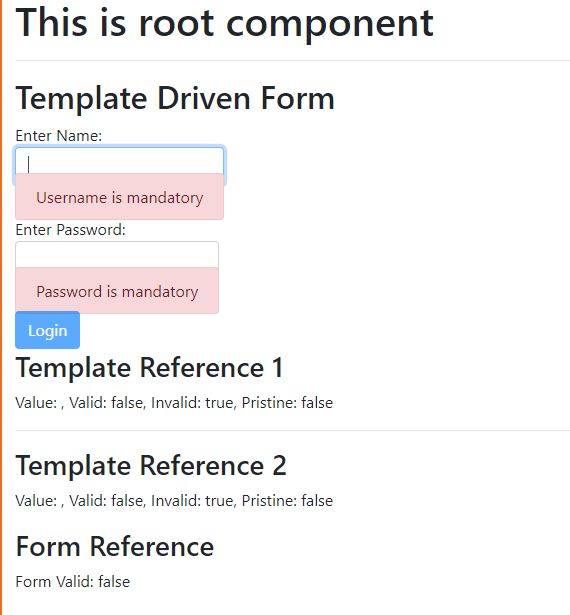
            <h3>Form Reference</h3>

            <p>Form Valid: {{userForm.valid}}</p>

        </div>

    </div>

</div>



Reactive Forms or Model Driven Forms

It will have more control over the form manipulation and its better for complex forms, it handles form data, validation everything in the typescript class so that your html will be neater.

You don’t have to use any template reference/ngModel, required apart from styling.

ReactiveForms gives you some inbuilt object like FormBuilder using which you can create an interactive form that is internally bound to the view template & component class

Note: ReactiveFormsModule must be imported in the app.module.ts to use the inbuild FormBuilder object.

FormBuilder object is automatically injected to the components, you don’t have to create them, angular supports constructor injection through it you can have parameter of FormBuilder.

app.module.ts



Now angular can create FormBuilder object which can be supplied to the component via constructor injection, the components must have a constructor() with FormBuilder parameter.

FormBuilder will give some functions like group() where you can group the form controls

How to create constructor with FormBuilder parameter

constructor(private form : FormBuilder) { }

How to create form controls?

loginForm = this.form.group({  
 user : [], pass : []  
});

user: [], It is an empty form without any values/validators

user : [‘test’], it will have a value test

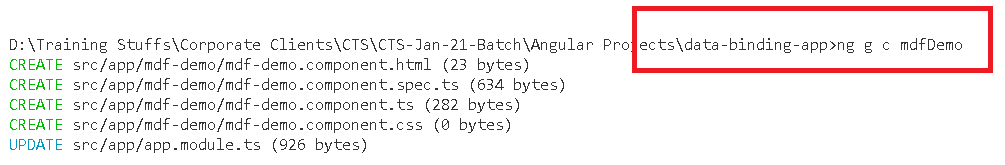
user : [‘’, Validators.required], will have a required validator

How to access the loginForm in html

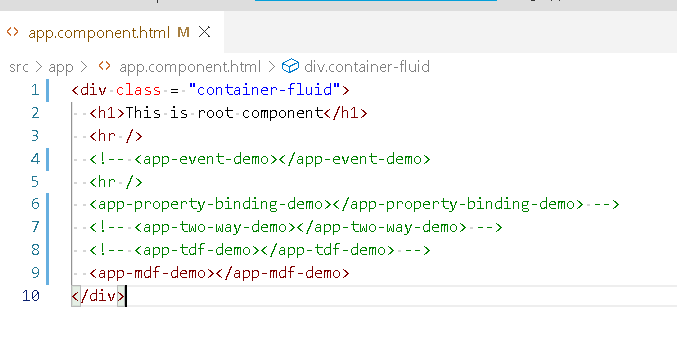
<form [formGroup] = “loginForm”>

<input formControlName = “user”>  
 <input formControlName = “pass”>  
</form>

Reactive Form Examples



Adding mdfDemo component to the app component



mdf-demo.component.ts

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

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

@Component({

  selector: 'app-mdf-demo',

  templateUrl: './mdf-demo.component.html',

  styleUrls: ['./mdf-demo.component.css']

})

export class MdfDemoComponent   {

  constructor(private form : FormBuilder) { }

  user = this.form.group({

    firstname : [],

    lastname : []

  });

  handleSubmit() {

    console.log(this.user.value);

    //this.user.reset();

    this.user.setValue({firstname :"", lastname: "Hello"})

  }

}

mdf.demo.component.html

<div>

    <h2>MDF Demo!</h2>

    <form [formGroup] = "user" (ngSubmit) = "handleSubmit()">

        <label>

            Firstname <input type = "text" formControlName = "firstname">

        </label>

        <br />

        <label>

            Lastname <input type = "text" formControlName = "lastname">

        </label>

        <br />

        <input type = "submit" value = "submit">

    </form>

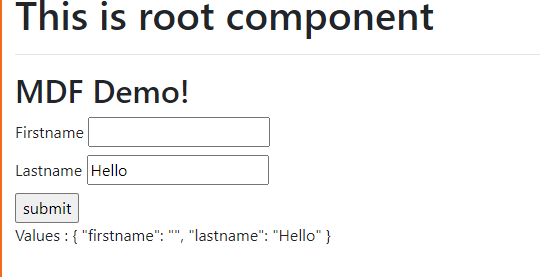
    <div>

        Values : {{user.value | json}}

    </div>

</div>

Output:



How to do validations in Model Driven Form

user = this.form.group({

firstname : [‘defaultValue’, Validators.required]  
});

mdf.comonent.ts

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

import { FormBuilder, Validators } from '@angular/forms';

@Component({

  selector: 'app-mdf-demo',

  templateUrl: './mdf-demo.component.html',

  styleUrls: ['./mdf-demo.component.css']

})

export class MdfDemoComponent   {

  constructor(private form : FormBuilder) { }

  user = this.form.group({

    firstname : ['', Validators.required],

    lastname : ['', Validators.required]

  });

  handleSubmit() {

    console.log(this.user.value);

    //this.user.reset();

    this.user.setValue({firstname :"", lastname: "Hello"})

  }

}

mdf.component.html

<div>

    <h2>MDF Demo!</h2>

    <form [formGroup] = "user" (ngSubmit) = "handleSubmit()">

        <label>

            Firstname <input type = "text" formControlName = "firstname">

        </label>

        <br />

        <label>

            Lastname <input type = "text" formControlName = "lastname">

        </label>

        <br />

        <input type = "submit" value = "submit" [disabled] = "user.invalid">

    </form>

    <div>

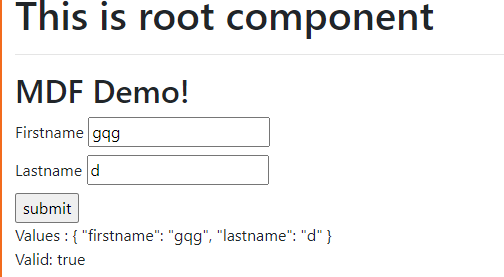
        Values : {{user.value | json}}<br />

        Valid: {{user.valid}}

    </div>

</div>

Output:



>> ng new employee-app

>> npm install bootrap --save

Add the imports in the styles.css

@import '~bootstrap/dist/css/bootstrap.css'

Angular Services:

These are set of class which are reusable by components, you can write business logics in the service and can be reused by multiple components.

How to create service

You can create service by creating a class and mentioning @Injectable() on top of the class

@Injectable({})  
export class EmpService {   
 login() {…}  
 store() {…}  
 fetch() {…}  
}

@Injectable(): marks the class instance to be injectable type so that angular creates such objects and supplies the objects to the components via constructor injection.

Angular provides a command to generate the service

>> ng generate service servie-name

>> ng g s service-name

Let us create one fake.ts file where you can keep some dummy data

export const employees : any = [

    {"id" : 100, "name":"Alex", "dob" : "1998-10-20"},

    {"id" : 200, "name":"Bruce", "dob" : "1999-10-20"},

    {"id" : 300, "name":"Charles", "dob" : "2000-10-26"},

    {"id" : 400, "name":"David", "dob" : "2001-11-10"},

    {"id" : 500, "name":"Edward", "dob" : "1996-10-20"}

];

employee.service.ts

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

import { employees } from './fake'

@Injectable({

  providedIn: 'root'

})

export class EmpService {

  constructor() { }

  fetchEmployees() : any {

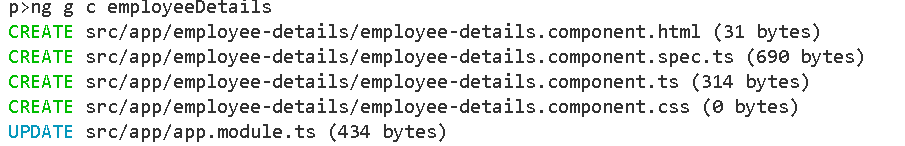
    return employees;

  }

}

Now fetchEmployees() must be called by component

>> ng g c employeeDetails



Now component must call fetchEmployees hence it needs the service object

employee-details.component.ts

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

import { EmpService } from '../emp.service';

@Component({

  selector: 'app-employee-details',

  templateUrl: './employee-details.component.html',

  styleUrls: ['./employee-details.component.css']

})

export class EmployeeDetailsComponent implements OnInit {

  emps : undefined | any = undefined;

  constructor(private service : EmpService) { }

  ngOnInit(): void {

  }

  fetchData() : void {

    this.emps = this.service.fetchEmployees();

  }

}

Now you can show the employee records in the employee-detials.component.html

<div>

    <h2>All employee records</h2>

    <button (click) = "fetchData()">Load</button>

    <div \*ngFor = "let e of emps">

        Id : {{e.id}}, Name : {{e.name}} and Dob: {{e.dob}}

    </div>

</div>

Add this emp-details component to the root component

<div>

  <h1>This is employee app</h1>

  <hr />

  <app-employee-details></app-employee-details>

</div>

HttpClient: It is an instance to call the backend webservices using http methods like get(), post(), put() delete() and etc.

Angular allows this object to be available in your application with the help HttpClientModule, this module must be part of Your app.module.ts imports section

@NgModule({  
 imports: [HttpClientModule]  
})

Angular will maintain HttpClient instance & supplies to different services

Mention the HttpClientModule in the imports of app.module.ts

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

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

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

import { EmployeeDetailsComponent } from './employee-details/employee-details.component';

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

@NgModule({

  declarations: [

    AppComponent,

    EmployeeDetailsComponent

  ],

  imports: [

    BrowserModule, HttpClientModule

  ],

  providers: [],

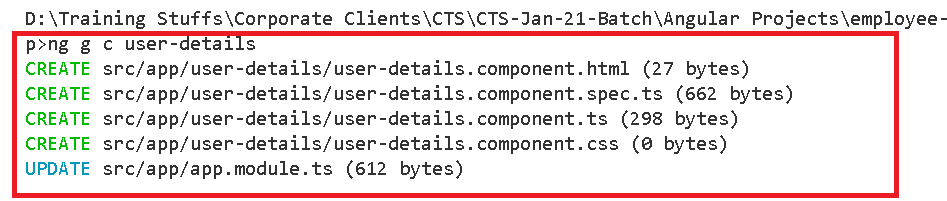
  bootstrap: [AppComponent]

})

export class AppModule { }

We will try to access the <https://jsonplaceholder.typicode.com/users>

Webservice using HttpClient hence you can create one component that will render the users coming from the above uri



We need to get the json data from the url

<https://jsonplaceholder.typicode.com/users>

For that we need HttpClient object,

But where to use HttpClient

You have to use HttpClient in the Service not in the Component.

You will write

constructor(private http : HttpClient) { }

HttpClient has methods all of them returns Observable type.

Observable: it is an instance that stores the response from the webservice

i.e., http.post(url, data) : Observable<any>

http.get(url): Observable<any>

emp.service.ts

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

import { employees } from './fake';

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EmpService {

  constructor(private http : HttpClient) { }

  fetchEmployees() : any {

    return employees;

  }

  fetchUsers() : Observable<any> {

    let uri = "https://jsonplaceholder.typicode.com/users";

    return this.http.get(uri);

  }

}

User-details.component.ts

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

import { EmpService } from '../emp.service';

@Component({

  selector: 'app-user-details',

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

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

})

export class UserDetailsComponent implements OnInit {

  users : any | undefined = undefined;

  constructor(private service : EmpService) { }

  ngOnInit(): void {

  }

  fetchUsersData() : void {

    this.service.fetchUsers().subscribe((s) => this.users = s);

  }

}

User-details.component.html

<div>

    <h2>This is user data coming from the backend</h2>

    <button (click) = "fetchUsersData()">Load</button>

    <div>

        {{users | json}}

    </div>

</div>

App.component.html

<div>

  <h1>This is employee app</h1>

  <hr />

  <app-employee-details></app-employee-details>

  <hr />

  <app-user-details></app-user-details>

</div>

Calling other methods of HttpClient like post(), put(), delete()

Using json-server library to create the fake webservices

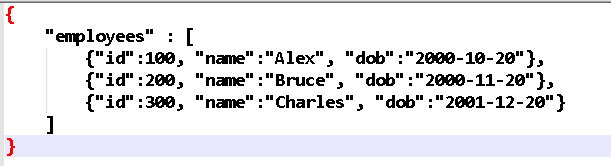
How does this json-server

It will have a json file in which you can have json data with some keys, these keys acts like url’s

>> npm install -g json-server

The above library allows you to create a fake webservice and you can call any of the http methods like post, put, delete, get and so on.

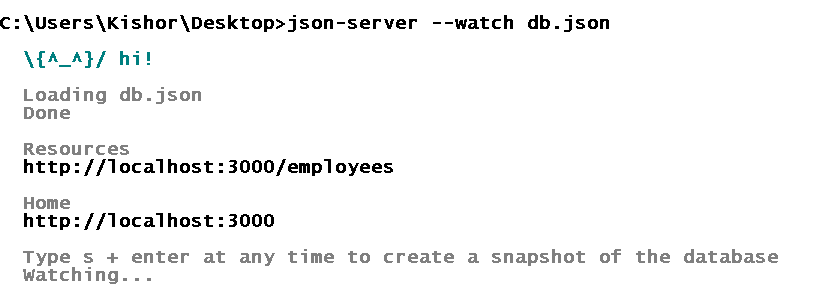
Create one file db.json



Now you can load this db.json using

>> json-server --watch db.json

It will load this file and makes it as a webservice so that you can call using http methods



Emp.service.ts

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

import { employees } from './fake';

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EmpService {

  constructor(private http : HttpClient) { }

  fetchEmployees() : Observable<any> {

    let uri = "http://localhost:3000/employees";

    //return employees;

    return this.http.get(uri);

  }

  fetchUsers() : Observable<any> {

    let uri = "https://jsonplaceholder.typicode.com/users";

    return this.http.get(uri);

  }

}

Emp-details.component.ts

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

import { EmpService } from '../emp.service';

@Component({

  selector: 'app-employee-details',

  templateUrl: './employee-details.component.html',

  styleUrls: ['./employee-details.component.css']

})

export class EmployeeDetailsComponent implements OnInit {

  emps : undefined | any = undefined;

  constructor(private service : EmpService) { }

  ngOnInit(): void {

  }

  fetchData() : void {

    //this.emps = this.service.fetchEmployees();

    // subscribe(success, error): it takes 2 parameters

    // success is for 200, error is for 400

    console.log('calling fetch data');

    this.service.fetchEmployees()

      .subscribe(

        (success) => console.log(success),

        (err) => console.log(err)

        );

  }

}

Employee-details.component.html

<div>

    <h2>All employee records</h2>

    <button (click) = "fetchData()">Load</button>

    <div \*ngFor = "let e of emps">

        Id : {{e.id}}, Name : {{e.name}} and Dob: {{e.dob}}

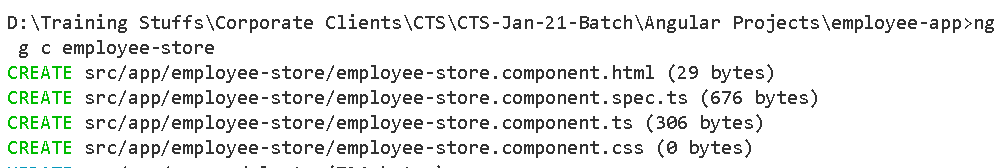
    </div>

</div>

How to store the data

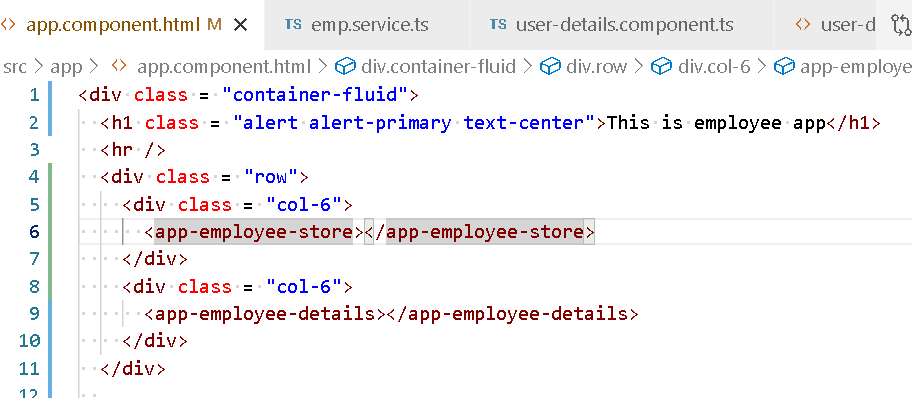
For storing the data we must use http.post(url, data), here the data would be json that reaches to the webservice.

>> ng g c employee-store



Add the employee-store and employee-details both in the root component and display using bootstrap row & col classes.

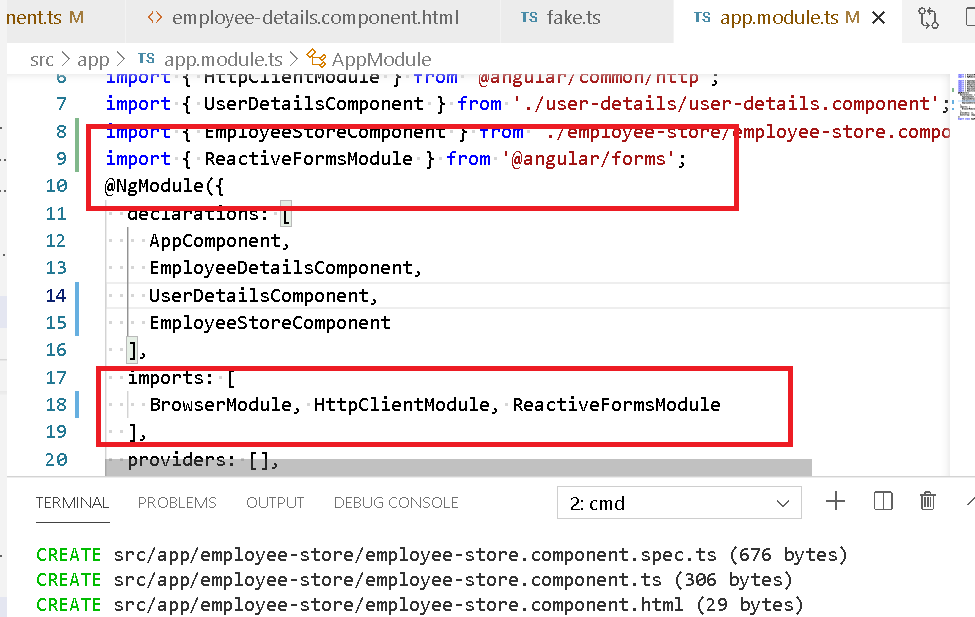
app.component.html



We need to create a form in the app-employee-store that component passes id, name & dob to the employees.json file using http.post(url, formData) to store in the employees.json file

You can use either TDF or MDF however you must use imports of FormsModule or ReactiveFormsModule.

We will use Model Driven Form so that first will update app.module.ts to have ReactiveFormsModule



We need to create a FormGroup in the employee-store.component.ts that is bind to the employee-store.component.html

Emp.service.ts

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

import { employees } from './fake';

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EmpService {

  baseUri = "http://localhost:3000/employees";

  constructor(private http : HttpClient) { }

  fetchEmployees() : Observable<any> {

    //return employees;

    return this.http.get(this.baseUri);

  }

  storeEmployee(employee : any) : Observable<any> {

    return this.http.post(this.baseUri, employee);

  }

  fetchUsers() : Observable<any> {

    let uri = "https://jsonplaceholder.typicode.com/users";

    return this.http.get(uri);

  }

}

The form group already sending the value in javascript object structure that can is internally converted to json structured by post, now you can call the storeEmployee from employee-store component

employee-store.component.ts

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

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

import { EmpService } from '../emp.service';

@Component({

  selector: 'app-employee-store',

  templateUrl: './employee-store.component.html',

  styleUrls: ['./employee-store.component.css']

})

export class EmployeeStoreComponent implements OnInit {

  constructor(private form : FormBuilder, private service : EmpService) { }

  employeeForm = this.form.group({

    id : [], name : [], dob : []

  });

  message : undefined | any = undefined;

  ngOnInit(): void {

  }

  handleSubmit() {

    this.service.storeEmployee(this.employeeForm.value)

    .subscribe((success) => {

      this.message = success;

      this.employeeForm.reset();

    });

  }

}

Employee-store.component.html

<div>

    <h2>Employee Form</h2>

    <div class = "form-group">

        <form [formGroup] = "employeeForm" (ngSubmit) = "handleSubmit()">

            <label>

                Enter id

                <input type = "text" formControlName = "id" class = "form-control">

            </label>

            <br />

            <label>

                Enter name

                <input type = "text" formControlName = "name" class = "form-control">

            </label>

            <br />

            <label>

                Dob

                <input type = "date" formControlName = "dob" class = "form-control">

            </label>

            <br />

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

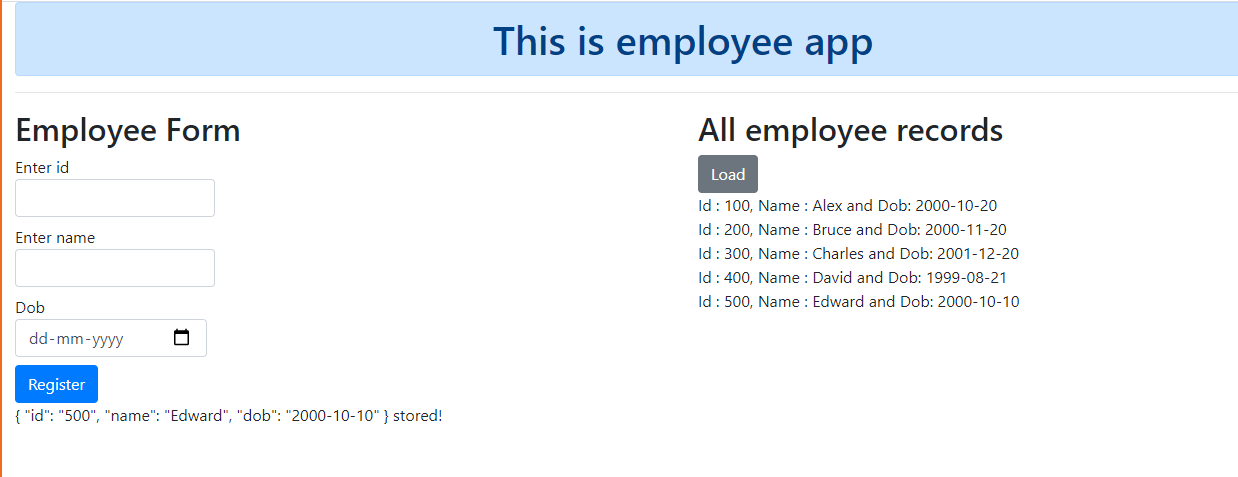
        </form>

        <div \*ngIf = "message">{{message | json}} stored!</div>

    </div>

</div>

Output:



Using Delete

Json server uses only id to delete, you can’t have some other properties apart form id, json server is a library which needs id mandatorily in the json file as a key, that is recognized as unique.

Emp.service.ts

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

import { employees } from './fake';

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EmpService {

  baseUri = "http://localhost:3000/employees";

  constructor(private http : HttpClient) { }

  fetchEmployees() : Observable<any> {

    //return employees;

    return this.http.get(this.baseUri);

  }

  storeEmployee(employee : any) : Observable<any> {

    return this.http.post(this.baseUri, employee);

  }

  deleteEmployee(empId : number) : Observable<any> {

    let uri = `${this.baseUri}/${empId}`; // employees/100

    return this.http.delete(uri);

  }

  fetchUsers() : Observable<any> {

    let uri = "https://jsonplaceholder.typicode.com/users";

    return this.http.get(uri);

  }

}

>> ng g c emp-delete

Add the emp-delete component in the root component

<div class = "container-fluid">

  <h1 class = "alert alert-primary text-center">This is employee app</h1>

  <hr />

  <div class = "row">

    <div class = "col-6">

      <app-employee-store></app-employee-store>

    </div>

    <div class = "col-6">

      <app-employee-details></app-employee-details>

    </div>

  </div>

  <div class = "row">

    <div class = "col-6">

      <app-emp-delete></app-emp-delete>

    </div>

  </div>

</div>

Emp-delete.component.ts

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

import { EmpService } from '../emp.service';

@Component({

  selector: 'app-emp-delete',

  templateUrl: './emp-delete.component.html',

  styleUrls: ['./emp-delete.component.css']

})

export class EmpDeleteComponent implements OnInit {

  message : undefined | string = undefined;

  constructor(private service : EmpService) { }

  ngOnInit(): void {

  }

  remove(id : number) : void {

    this.service.deleteEmployee(id)

    .subscribe(

      s => this.message = `${id} deleted successfully`,

      err => this.message = `Sorry ${id} not found`

      );

  }

}

Emp-delete.component.html

<div>

    <h3>Delete Employee</h3>

    <hr />

    <input type = "number" #tf class = "form-control">

    <button (click) = "remove(tf.value)" class = "btn btn-danger">Delete</button>

    <div \*ngIf = "message" class = "alert alert-danger">

        {{message}}

    </div>

</div>

Output:



Exercise:

* Update the dob by entering id and dob
* Fetch the employee details by id

Routers:

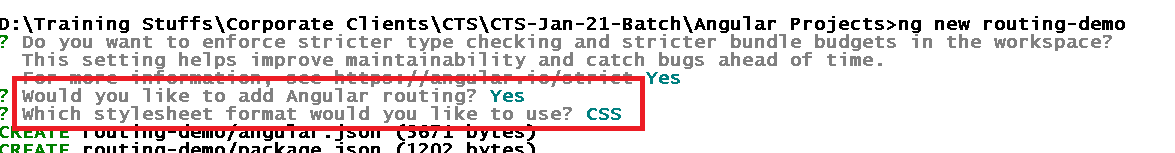
Routers are used to navigate from one component to another component, this is the backbone to develop single page application.

Router is not available by default you need to add RouterModule part of @angular/router in your application.

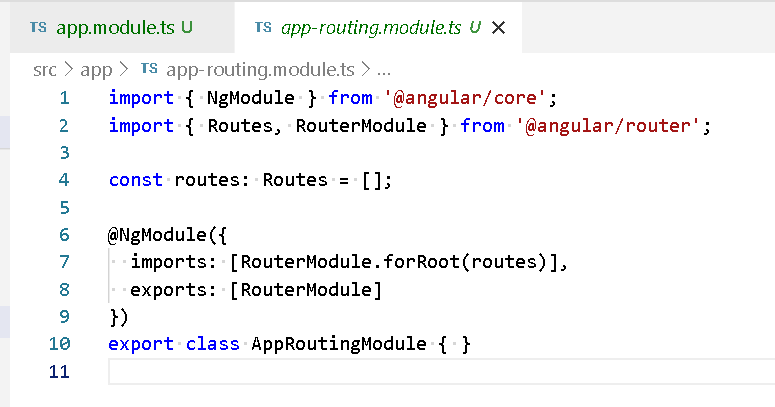
Router will use certain directives & objects inorder to load the components based on the action the user perform which are:

* Routes: it configures the component with the path url
* routerLink: it is a link with the path url to navigate in HTML template
* Router: it is going to provide the path url to navigate programmatically
* RouterModule: it is the module that helps router work, because it will have the complete routing configurations
* ActivatedRoute: it is used to extract the data shared while routing from one component to another
* <router-outlet>: it is the main element which is a place holder that dynamically loads the components based on the action

Create a new angular project and select router this time



When you select the routing feature the project will have one file app-routing.module.ts which will have Routes & RouterModule configured with @NgModule decorator, and it acts like a submodule and it will be added in the imports of the app.module.ts internally

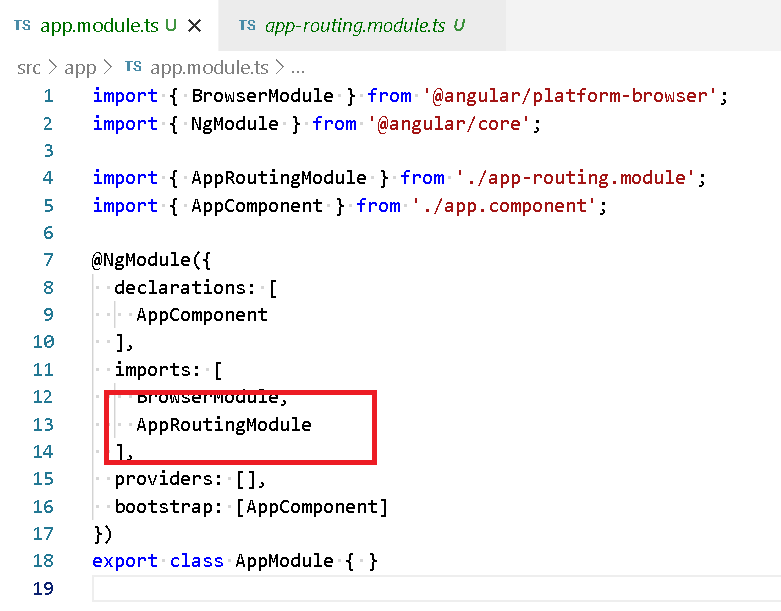


4th line will have a path & component configuration, but by default it will be empty.

7th line will know the complete routing configurations which helps in loading the component for the appropriate routes

10th line is the class with @NgModule it means it also a module but this class must be part of your root module, hence while you chose routing feature to your project you will be having this class in the imports section of AppModule

<router-outlet> will be provided by default in the app.component.html



Modify the app.component.html as below

<div>

  <h1>My simple routing app</h1>

  <router-outlet></router-outlet>

</div>

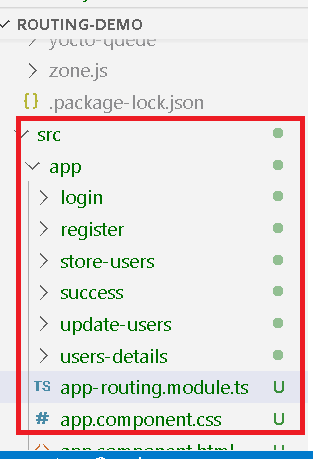
Note: By default the root component already as the router-outlet, which is the place holder for all the components

Initial setups

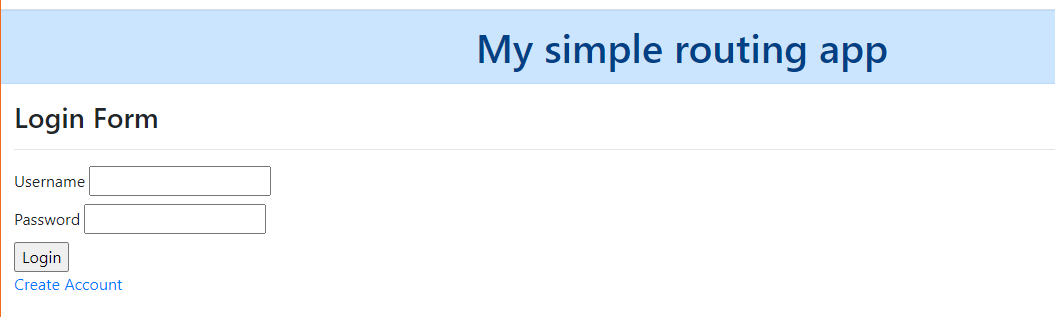
* Install bootstrap & add in the styles.css
* Create some components like login, register, user-details, update-users, store-users
* Create routerLink in the app-routing.module.ts

Create 5 components one by one

>> ng g c login  
>> ng g c register  
>> ng g c store-users  
>> ng g c user-details  
>> ng g c update-users  
>> ng g c success



Getting this Output:



app-routing.module.ts

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

import { Routes, RouterModule } from '@angular/router';

import { LoginComponent } from './login/login.component';

import { RegisterComponent } from './register/register.component';

import { SuccessComponent } from './success/success.component';

const routes: Routes = [

  {path : "", component : LoginComponent},

  {path : "login", component : LoginComponent},

  {path : "register", component : RegisterComponent},

  {path : "success", component : SuccessComponent}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

app.component.html

<div>

  <h1 class = "text-center alert alert-primary">My simple routing app</h1>

  <div class = "container-fluid">

    <router-outlet></router-outlet>

  </div>

</div>

Login.component.html

<div>

    <h3>Login Form</h3>

    <hr />

    <form>

        <label>

            Username <input type = "text">

        </label>

        <br />

        <label>

            Password <input type = "password">

        </label>

        <br />

        <input type = "submit" value = "Login">

    </form>

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

</div>

Register.component.html

<div>

    <h3>Registration Form</h3>

    <hr />

    <form>

        <label>

            Firstname <input type = "text">

        </label>

        <br />

        <label>

            Lastname <input type = "text">

        </label>

        <br />

        <label>

            Password <input type = "password">

        </label>

        <br />

        <input type = "submit" value = "Register">

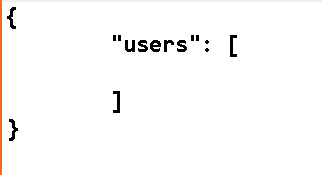
    </form>

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

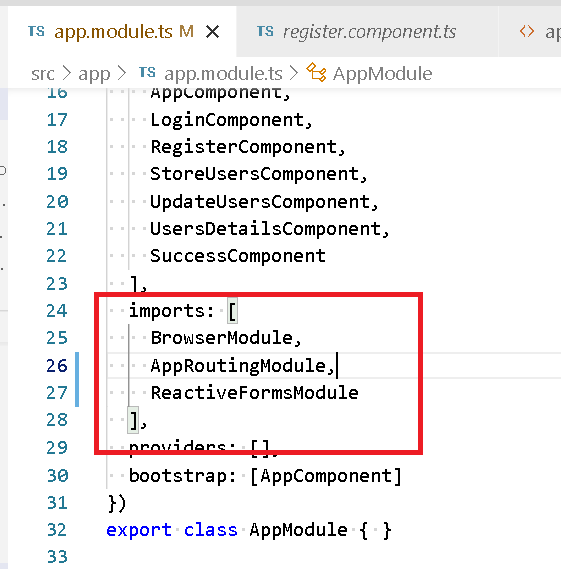
</div>

Creating the form group in the register component

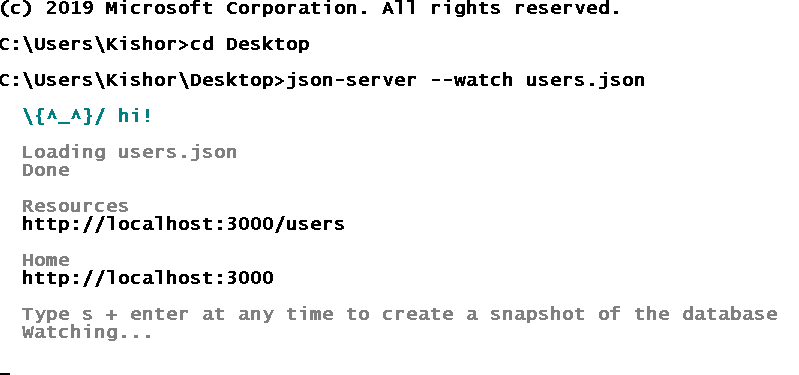
First we will create users.json file which will maintain user information’s like id, name, dob, phone



Add the ReactiveFormsModule in the AppModule

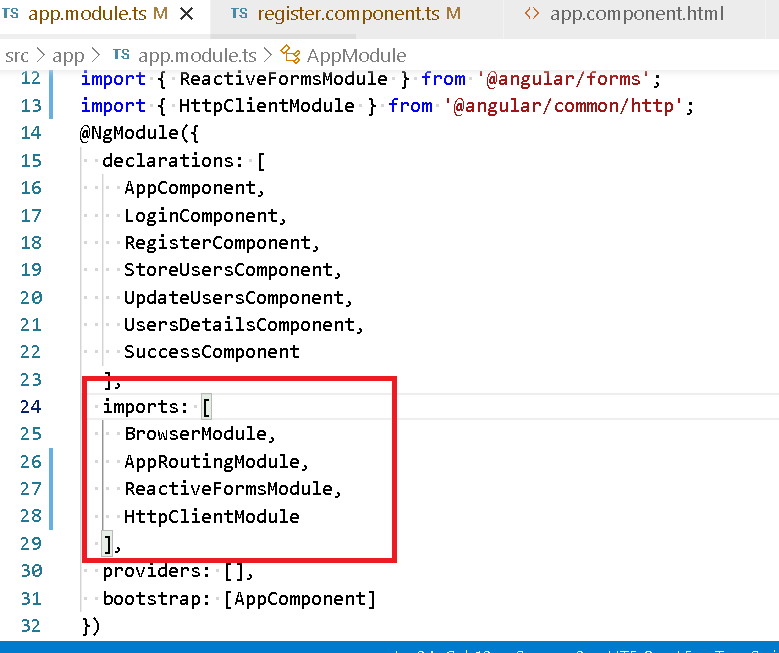


Start the json server to load users.json

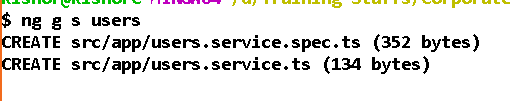


Create the service to perform store, update, delete & fetch operations using HttpClient

Add the HttpClientModule in the app.module.ts



>> ng g s users



users.service.ts

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

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class UsersService {

  baseUri : string = 'http://localhost:3000/users';

  constructor(private httpClient : HttpClient) { }

  storeUsers(userForm : any) : Observable<any> {

    return this.httpClient.post(this.baseUri, userForm);

  }

  fetch(id : number) : Observable<any> {

    return this.httpClient.get(`${this.baseUri}/${id}`);

  }

  fetchUsers() : Observable<any> {

    return this.httpClient.get(this.baseUri);

  }

}

Create the FormBuilder to handle the form data & once the data is stored successfully you can navigate to login component, here you need to do a programmatic navigation.

register.component.ts

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

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

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

import { UsersService } from '../users.service';

@Component({

  selector: 'app-register',

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

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

})

export class RegisterComponent implements OnInit {

  constructor(

    private form: FormBuilder,

    private service : UsersService,

    private router : Router

    ) { }

  userForm = this.form.group({

    id : [], name : [], dob : [], phone : []

  });

  ngOnInit(): void {

  }

  store() : void{

    this.service.storeUsers(this.userForm.value)

    .subscribe((success) => {

      this.router.navigate(["/login"])

    });

  }

}

register.component.html

<div>

    <h3>Registration Form</h3>

    <hr />

    <form [formGroup] = "userForm" (ngSubmit) = "store()">

        <label>

            Id <input type = "number" formControlName = "id">

        </label>

        <br />

        <label>

            Name <input type = "text" formControlName = "name">

        </label>

        <br />

        <label>

            Phone <input type = "text" formControlName = "phone">

        </label>

        <br />

        <label>

            dob <input type = "date" formControlName = "dob">

        </label>

        <br />

        <input type = "submit" value = "Register">

    </form>

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

</div>

app-routing.module.ts

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

import { Routes, RouterModule } from '@angular/router';

import { LoginComponent } from './login/login.component';

import { RegisterComponent } from './register/register.component';

import { SuccessComponent } from './success/success.component';

const routes: Routes = [

  {path : "", component : LoginComponent},

  {path : "login", component : LoginComponent},

  {path : "register", component : RegisterComponent},

  {path : "success/:userId", component : SuccessComponent}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

The path : success/:userId, means you can navigate to the success component using success/100, success/200, success/300 and so on, :userId is the path variable which can take dynamic path.

login.component.ts

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

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

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

import { UsersService } from '../users.service';

@Component({

  selector: 'app-login',

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

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

})

export class LoginComponent implements OnInit {

  constructor(

    private form : FormBuilder,

    private service: UsersService,

    private router : Router

    ) { }

  loginForm = this.form.group({

    id : [], password : []

  });

  ngOnInit(): void {

  }

  authenticate() : void {

    let id = this.loginForm.value.id;

    if(this.loginForm.value.password == 'Admin') {

      this.service.fetch(id).subscribe(

        success => {

          this.router.navigate(["/success", id]);

        }, err => {

          alert('Invalid Credentials');

          this.loginForm.reset();

          this.router.navigate(["/login"]);

        }

      );

    } else {

      alert('Invalid Credentials');

      this.loginForm.reset();

      this.router.navigate(["/login"]);

    }

  }

}

In the above example we are trying to check the hard coded password, however the id you enter is queried from the webservice.

Router: it is an instance to do a programmatic navigation

login.component.html

<div>

    <h3>Login Form</h3>

    <hr />

    <form [formGroup] = "loginForm" (ngSubmit) = "authenticate()">

        <label>

            Id <input type = "number" formControlName = "id">

        </label>

        <br />

        <label>

            Password <input type = "text" formControlName = "password">

        </label>

        <br />

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

    </form>

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

</div>

Now success component needs to extract the path parameter i.e., success/:userId, this path can have any value for :userId, hence that must be extracted so that success component can send that id to webservice and get the user information.

Suppose:

success/1 -> then success component must show user information of 1

success/2 -> then success component must show user information of 2

And so on.

success.component.ts

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

import { ActivatedRoute, Params } from '@angular/router';

import { UsersService } from '../users.service';

@Component({

  selector: 'app-success',

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

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

})

export class SuccessComponent implements OnInit {

  user : any | undefined = undefined;

  constructor(

    private activatedRoute : ActivatedRoute,

    private service : UsersService

    ) { }

  ngOnInit(): void {

   this.activatedRoute.params.subscribe((p : Params) => {

     let id = p.userId;

     this.service.fetch(id).subscribe(success => this.user = success);

   });

  }

}

success.component.html

<div>

    <h2>User Profile</h2><hr />

    <div \*ngIf = "user">

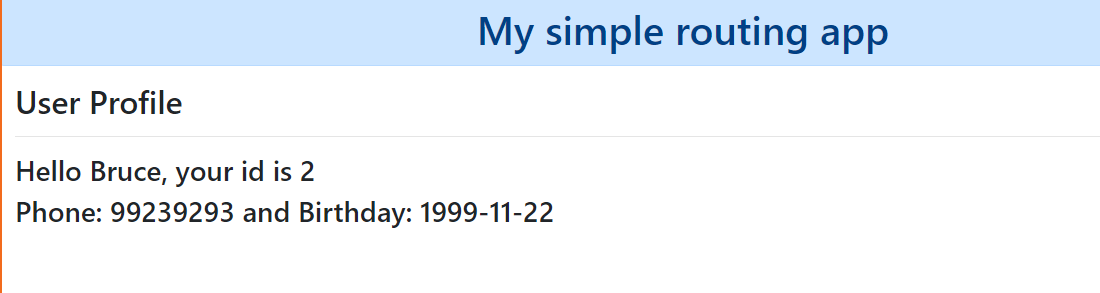
        <h3>Hello {{user.name}}, your id is {{user.id}}</h3>

        <h3>Phone: {{user.phone}} and Birthday: {{user.dob}}</h3>

    </div>

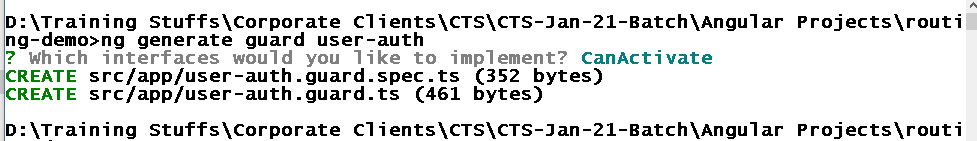
</div>

Output:



The above program will allow even non authenticated user also to enter the success component hence you must guard the routing, so that you can prevent non-authenticated users to enter without authenticated, for that we have auth-guards in angular

>> ng generate guard user-auth



CanActivate is an interface with that has to be implemented which has a method called canActivate() that is automatically called when you route to the particular component that uses this Guard

How to add the guard while routing

You need to mention the class in the routes using the property canActivate

user-auth.guard.ts

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

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree, Router } from '@angular/router';

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class UserAuthGuard implements CanActivate {

  // using this we can navigate on error

  constructor(private router : Router) { }

  canActivate(route: ActivatedRouteSnapshot,

    state: RouterStateSnapshot

    ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree

  {

      if(route.params.userId == sessionStorage.getItem('userKey'))

        return true;

      else {

        alert('You need to login first');

        this.router.navigate(['/login'])

        return false;

      }

  }

}

Mentioning the UserAuthGuard while routing

app-routing.module.ts

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

import { Routes, RouterModule } from '@angular/router';

import { LoginComponent } from './login/login.component';

import { RegisterComponent } from './register/register.component';

import { SuccessComponent } from './success/success.component';

import { UserAuthGuard } from './user-auth.guard';

const routes: Routes = [

  {path : "", component : LoginComponent},

  {path : "login", component : LoginComponent},

  {path : "register", component : RegisterComponent},

  {path : "success/:userId",

    component : SuccessComponent, canActivate: [UserAuthGuard]}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

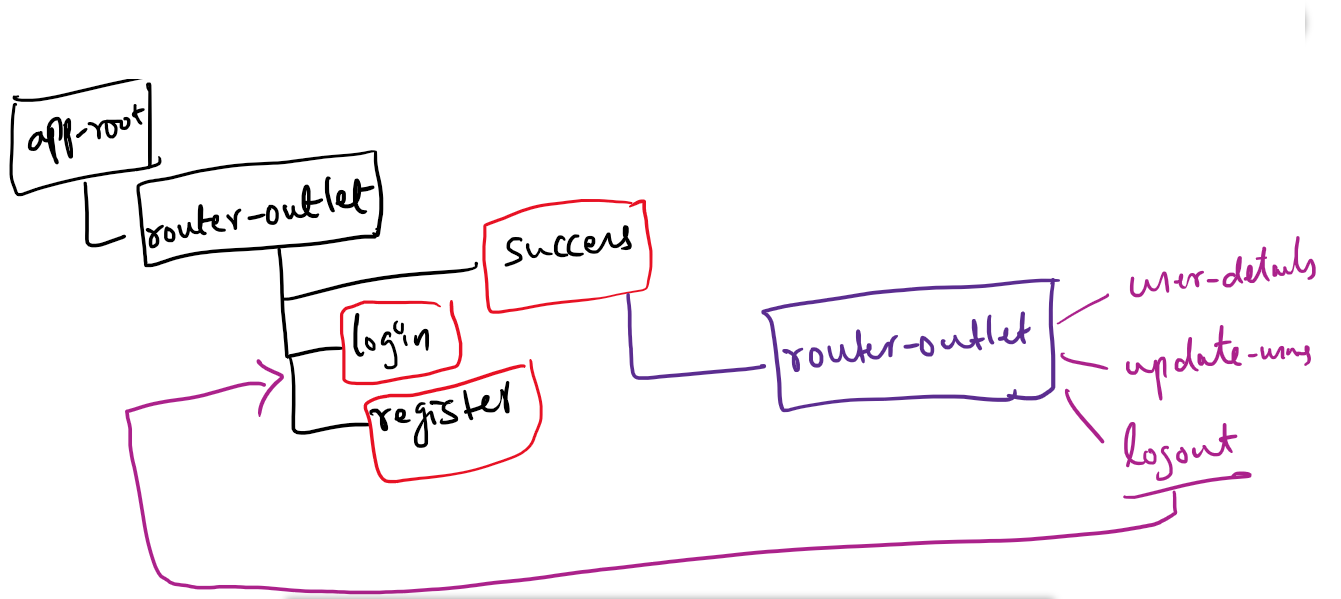
  exports: [RouterModule]

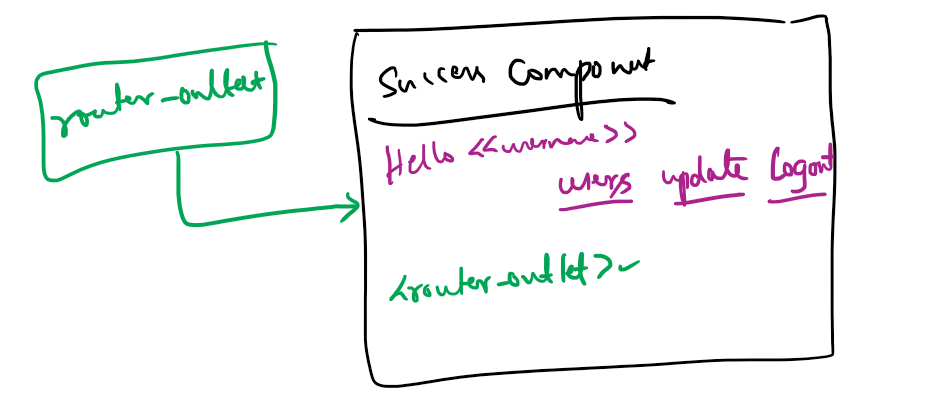
})

export class AppRoutingModule { }

Exercise:

Add the <router-outlet> in the success component and load some components like user-details, update-users with some links





Testing the angular application

Testing is nothing checking the application is working as expected, we have a karma automation tool in angular to run the unit test cases.

You can write unit test cases in .spec.ts, the karma tool will automatically recognize the files with .spec.ts extension & runs the test cases written in it.

How to run the test cases

We need to use ng test command to run the test cases

ex:

export class DemoComponent {   
 name : string = undefined;  
   
 handleClick() {   
 // you change the value of name  
 }  
}

Above component will change name when handleClick is called, for that you can have a unit test case written so that you can call handleClick and expect the name to be changed.

How to run the unit test case for the DemoComponent

In the .spec.ts file you can write unit test cases for the Demo Component

describe(‘collection of test cases-suites’,   
 function() {  
 it(‘unit test case description’, function() {  
 })  
 }

);

The describe() is a function that acts like a test suite that is group of test cases and it will have 2 parameters

1st parameter is the description & 2nd parameter is the anonymous function or arrow function which will have multiple unit test cases.

describe(‘…’, () => {  
 it(‘…’, () => {});  
 it(‘…’, () => {});  
});

Spy: You can also mock the objects when you want fake instances instead of using the real instance, like faking router, services, httpclient and so on, in angular it called as spy.

Firstly, we can test the functions which are not part of the component & later we can test the functions that are part of the component.

Simple.ts

export function greetings(name : string) {

    return "Welcome "+name;

}

export function incrementByOne(input : number) {

    return input + 1;

}

// greetings('Alex') -> Welcome Alex

// incrementByOne(20) -> 21

Simple.spec.ts

import { greetings, incrementByOne } from './simple';

// all the unit test cases are wrapped in a suite which is 'describe'

describe('Functions inside simple', () => {

    // unit test cases are written in the 'it'

    it('Check for the welcome message', () => {

        const result = greetings('Alex');

        expect(result).toContain('Alex');

    });

    // unit test case for incrementByOne

    it('Increment the number by 1', () => {

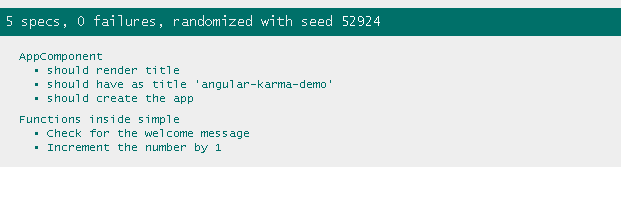
        const value = incrementByOne(1);

        expect(value).toBeGreaterThan(1)

    });

});

Output:



Testing the components

>> ng g c counter

Counter.component.ts

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

@Component({

  selector: 'app-counter',

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

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

})

export class CounterComponent  {

 counter : number = 0;

 handleClick() {

  this.counter++;

 }

}

Counter.component.spec.ts

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { CounterComponent } from './counter.component';

describe('CounterComponent', () => {

  let component: CounterComponent;

  let fixture: ComponentFixture<CounterComponent>;

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      declarations: [ CounterComponent ]

    })

    .compileComponents();

  });

  beforeEach(() => {

    fixture = TestBed.createComponent(CounterComponent);

    component = fixture.componentInstance;

    fixture.detectChanges();

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

  it('verify the counter value', () => {

    component = fixture.componentInstance;

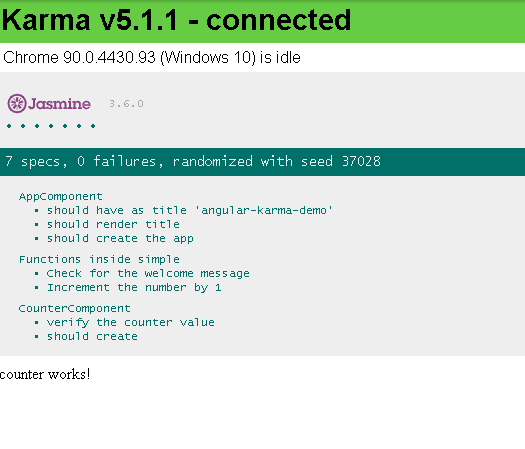
    component.handleClick();

    expect(component.counter).toBeGreaterThan(0);

  });

});

Output:



Exercise: You can test the html content also with the help of

 const compiled = fixture.nativeElement;

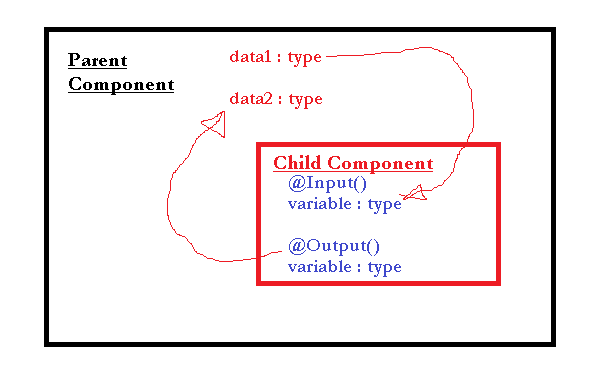
 expect(compiled.querySelector('p').textContent).toContain('some message');

@Input() & @Output() decorators:

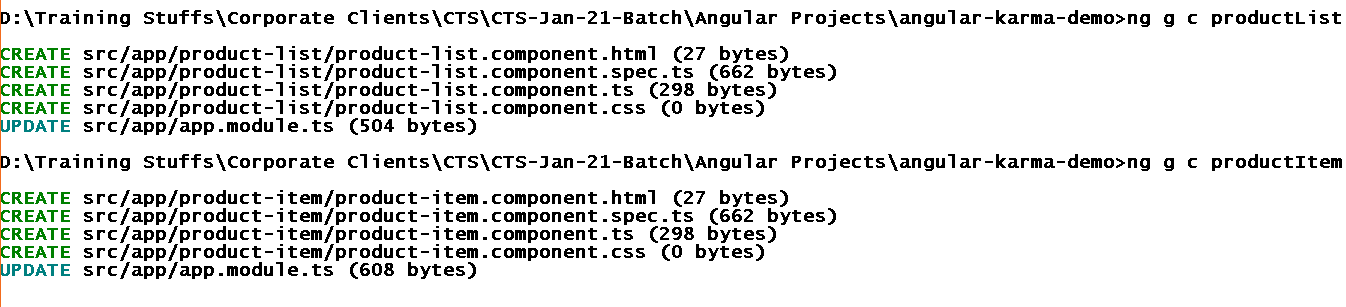
It is used to share the data between the components i.e., from parent to child and from child to parent

@Input() : it is used to share the data from parent to child

@Output() : it is used to share the data from child to parent



>> ng g c productList  
>> ng g c productItem



Will add the product-item to the product-list and product-list to the root component

app.component.html

<div>

  <h2>This is root component</h2> <hr />

  <app-product-list></app-product-list>

</div>

product-list.component.html

<div>

    <h2>This is product list component which is parent</h2>

    <hr />

    <app-product-item></app-product-item>

</div>

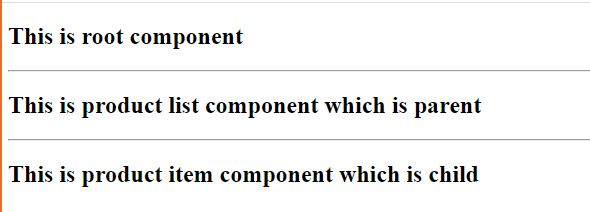
Here we are not sharing the data from root component to product list component instead we are sharing the data from product list to product item

product-item.component.html

<div>

    <h2>This is product item component which is child</h2>

</div>

Output before using @Input & @Output  


Now we will share the data from product list to the product item

We will have an array of products name in the product list and product item must display one item

Create @Input & @Output in the child component

product-item.component.ts

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

@Component({

  selector: 'app-product-item',

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

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

})

export class ProductItemComponent {

  // <app-product-item [product] = "value"></app-product-item>

  @Input()

  product : string | undefined = undefined;

  // <app-product-item (eventEmitter) = "parentProps = $event">

  @Output()

  eventEmitter : EventEmitter<number> = new EventEmitter<number>();

  handleClick() {

    this.eventEmitter.emit(1);

  }

}

product-item.component.html

<div>

    <h3>This is child component</h3>

    <p>{{ product }}</p>

    <button (click) = "handleClick()">Add</button>

</div>

Using the child component in the parent component

product-list.component.ts

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

@Component({

  selector: 'app-product-list',

  templateUrl: './product-list.component.html',

  styleUrls: ['./product-list.component.css']

})

export class ProductListComponent  {

  productsList : string[] = ["Laptops", "Mobiles", "Notepads"];

  parentCounter : number = 0;

  takeAllClicks(input : number) {

    this.parentCounter = this.parentCounter + input;

  }

}

product-list.component.html

<div>

    <h2>This is product list component which is parent</h2>

    <h3>Parent Counter: {{ parentCounter }}</h3>

    <hr />

    <div \*ngFor = "let p of productsList">

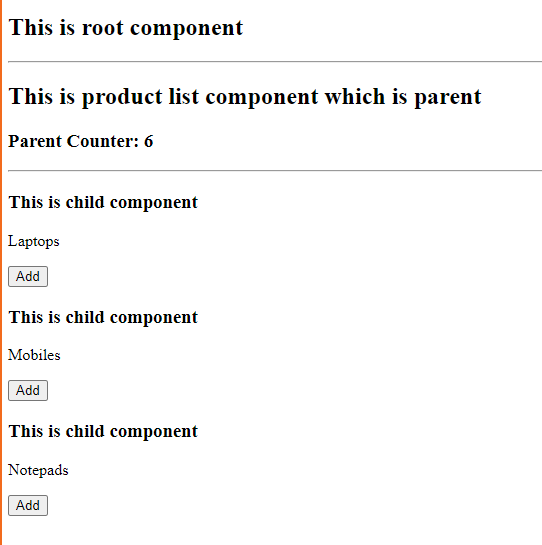
        <app-product-item

        [product] = "p" (eventEmitter) = "takeAllClicks($event)"></app-product-item>

    </div>

</div>

Output:



Note:

Here we are sharing simple value, however you can also share complex values like object or arrays and etc.

Spring ReST webservices

ReST stands for Representational State Transfer, it is used to share the data between applications written in different languages.

Ex: Banking applications are accessed by Google Pay, PayTM, Phone Pe, Amazon, Flipkart, IRCTC and so on

Ex: Google Maps are accessed by Swiggy, Zomato, OLA, Uber.

There are two types of webservices

1. SOAP based webservices (old)
2. ReSTful webservices (new)

SOAP stands for Simple Object Access Protocol, here the data structure will be always XML, which may be heavy weight to parse and also end applications use interfaces to call the business logics/remote webservices, SOAP webservices use SOA architecture, which needs Service Requester, Service Broker and Service Producer

REST stands for Representational State Transfer, here the data structure can be JSON, XML, Text, CSV and so, here the client can get the data in any format which is easier for it to parse, here the end applications use HTTP methods and uri to communicate, REST uses client-server architecture which needs only URI’s and HTTP methods.

RESTful webservices can be implemented in any languages like Javascript, Java, C#, Python and so on.

In Java we have Spring Boot REST to implement the webservices

Pre-requisites

* Spring Framework
* Spring Annotations
* Spring Boot

Spring Boot project can be created using maven project / spring initializr

Spring Boot:

It allows you to quickly create the spring applications without any external configurations, it has many features

* Auto configurations
* No xml required
* Embedded server for web applications
* Creates the dependencies based on the libraries you add
* Creates the production ready application
* Accelerates the development

Spring REST:

You can create spring restful webservices using @RestController annotation, it has to be added in the controller layer

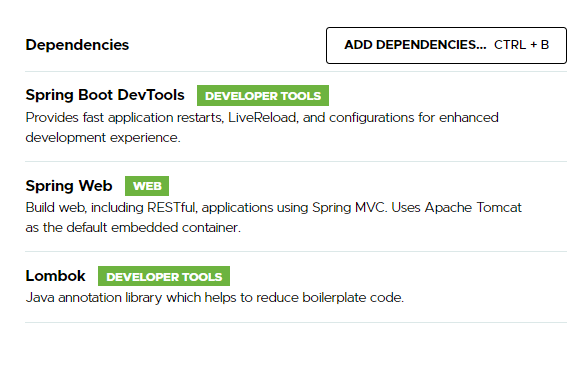
MVC pattern:

Model layer: Will have Java beans, DAO’s, Services

Controller layer: Will have @Controller, @RestController

View layer: Will have JSP’s or HTML or it could be JSON/XML for different applications written in different languages

Create a project in spring initializr



Some of the annotations we are going to use:

@SpringBootApplication: It launches the spring application & also does the auto-configuration based on the library you add.

@RestController: It is written on top of the class to create rest services with URI’s http methods, data representation and so on.

@RequestMapping: It is used to mention the uri’s, http methods

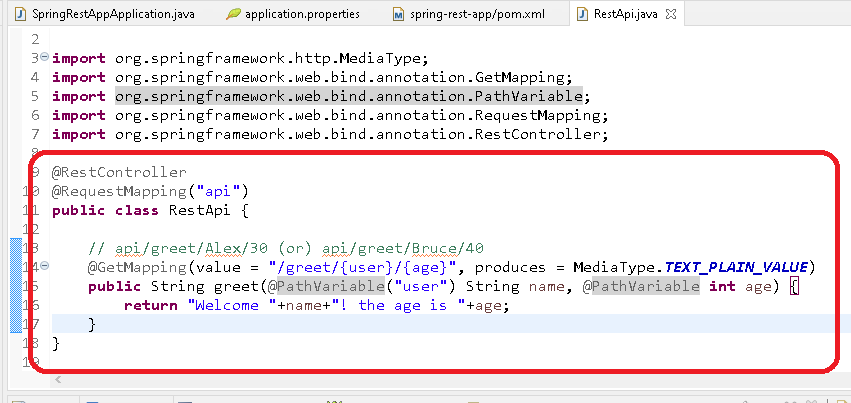
@GetMapping: It is similar to @RequestMapping but short cut for HTTP get

@PostMapping: It is similar to @RequestMapping but short cut for HTTP post, same way have short cut for Put and Delete they are @PutMapping, DeleteMapping

@RequestBody: It is used to extract the data from request body that will have JSON/XML

@PathVariable: It is used to extract the data from the dynamic path

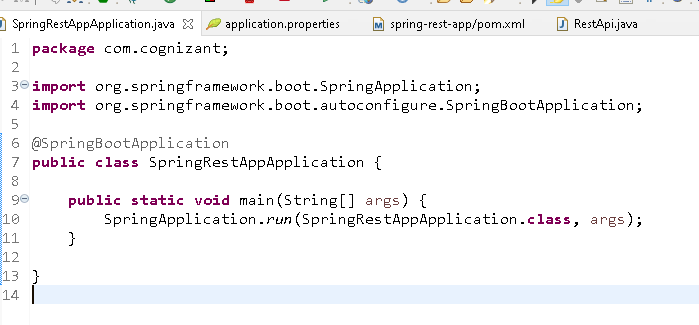
RestApi.java



@GetMapping is the short cut for @RequestMapping(method = RequestMethod.GET), produces is not mandatory because spring boot takes care of converting the return type to corresponding compatible structure, like if String it converts to text/plain, if it is Java object it converts to JSON, produces is necessary when you want to mention other structure

@PathVariable is used to extract the path parameter, the PathVariable doesn’t need to mention the name if the path parameter name and method parameter name are same i.e., /{user} is mapping to name it needs @PathVariable(“user”) however /{age} is mapping to age which doesn’t need @PathVariable(“age”), it just needs @PathVariable

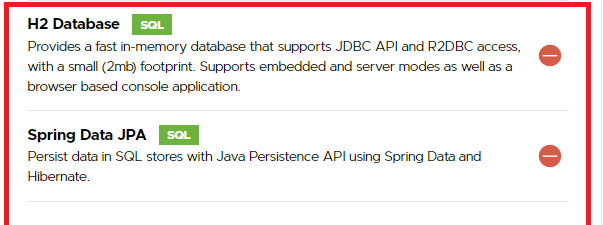
SpringRestAppApplication.java



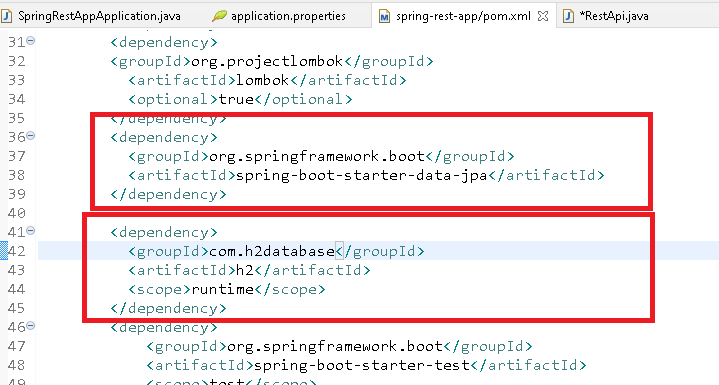
Using Spring Data JPA with H2 database

H2 is a in-memory database which is used when you don’t want to use the real-database, it can use all the ORM operations what the real database uses, but the H2 database will store the data until the next restart of the server.

Adding h2 database & spring data jpa library



pom.xml



Now configuring h2 database url, username, password, drivername in the application.properties

application.properties

server.port = 8081

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

Employee.java

**package** com.cognizant.model.entities;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.Table;

**import** lombok.Getter;

**import** lombok.Setter;

**import** lombok.ToString;

@Entity

@Table(name = "employee")

@ToString

@Getter

@Setter

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** **int** id;

**private** String name;

**private** **double** salary;

}

EmployeeRepository.java

**package** com.cognizant.model.dao;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.cognizant.model.entities.Employee;

**public** **interface** EmployeeRepository **extends** JpaRepository<Employee, Integer>{

}

Now you can auto-wire the EmployeeRepository in the @Service class, firstly we will create interface for the service & then implement it in the @Service class

EmployeeService.java

**package** com.cognizant.model.service;

**import** java.util.List;

**import** com.cognizant.model.entities.Employee;

**public** **interface** EmployeeService {

**public** List<Employee> fetchEmployees();

**public** Employee saveEmployee(Employee employee);

**public** Employee fetchEmployee(**int** id);

}

EmployeeServiceImpl.java

**package** com.cognizant.model.service;

**import** java.util.List;

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

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

**import** com.cognizant.model.dao.EmployeeRepository;

**import** com.cognizant.model.entities.Employee;

@Service

**public** **class** EmployeeServiceImpl **implements** EmployeeService {

@Autowired

**private** EmployeeRepository employeeDao;

@Override

**public** List<Employee> fetchEmployees() {

**return** employeeDao.findAll();

}

@Override

**public** Employee saveEmployee(Employee employee) {

Employee created = employeeDao.save(employee);

**return** created;

}

@Override

**public** Employee fetchEmployee(**int** id) {

Employee employee = employeeDao.findById(id).get();

**return** employee;

}

}

RestApi.java

**package** com.cognizant.controller;

**import** java.util.List;

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

**import** org.springframework.http.MediaType;

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

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

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

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

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

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

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.service.EmployeeService;

@RestController

@RequestMapping("api")

**public** **class** RestApi {

@Autowired

**private** EmployeeService service;

@PostMapping(path = "employees", consumes = MediaType.***APPLICATION\_JSON\_VALUE***,

produces = MediaType.***APPLICATION\_JSON\_VALUE***)

**public** Employee storeApi(@RequestBody Employee emp) {

Employee createdEmployee = service.saveEmployee(emp);

**return** createdEmployee;

}

@GetMapping(path = "employees", produces = MediaType.***APPLICATION\_JSON\_VALUE***)

**public** List<Employee> getEmployeesApi() {

**return** service.fetchEmployees();

}

@GetMapping(path = "employees/{id}", produces = MediaType.***APPLICATION\_JSON\_VALUE***)

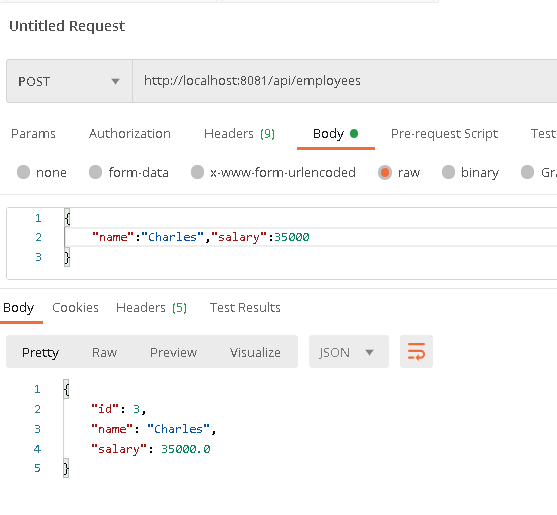
**public** Employee getEmployeesByIdApi(@PathVariable **int** id) {

**return** service.fetchEmployee(id);

}

}

Output:



Content Negotiation:

The client applications can send the request in the request header to obtain a different response like json, xml, text and also client applications can send the request with different data like json, xml, text and so on, so the webservices would able to accept any type of content also produce any type of content, but the webservices should have that configuration.

Consuming & Producing XML

Your java class / model must use @XmlRootElement which is JAXB feature that takes care of converting java to xml and vice versa.

Once you mention this @XmlRootElement the spring annotation that could able to either produce or consume data must specify the content type like JSON, XML and so on

Employee.java

**package** com.cognizant.model.entities;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.Table;

**import** javax.xml.bind.annotation.XmlRootElement;

**import** lombok.Getter;

**import** lombok.Setter;

**import** lombok.ToString;

@XmlRootElement

@Entity

@Table(name = "employee")

@ToString

@Getter

@Setter

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** **int** id;

**private** String name;

**private** **double** salary;

}

Now the REST apis must use XML format to either consume/produce

Note: You can use both JSON and XML but if you want to get a specific format you must specify in the request header using some http header properties like accept-type / content-type

Accept-type: it is used when client needs a response in a specific format

Content-type: it is used when client is sending the data in a specific format to the server

RestApi.java

**package** com.cognizant.controller;

**import** java.util.List;

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

**import** org.springframework.http.MediaType;

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

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

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

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

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

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

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.service.EmployeeService;

@RestController

@RequestMapping("api")

**public** **class** RestApi {

@Autowired

**private** EmployeeService service;

@PostMapping(path = "employees",

consumes = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE*** },

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** Employee storeApi(@RequestBody Employee emp) {

Employee createdEmployee = service.saveEmployee(emp);

**return** createdEmployee;

}

@GetMapping(path = "employees",

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** List<Employee> getEmployeesApi() {

**return** service.fetchEmployees();

}

@GetMapping(path = "employees/{id}",

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

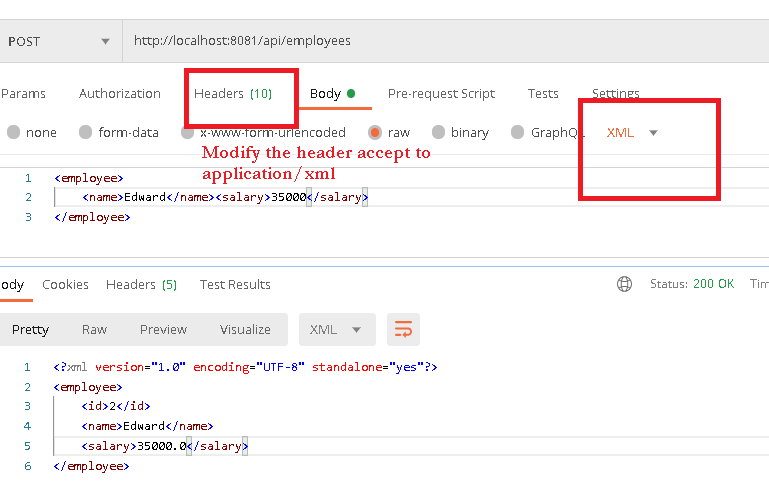
**public** Employee getEmployeesByIdApi(@PathVariable **int** id) {

**return** service.fetchEmployee(id);

}

}

Output:



Http status codes

These are response status codes when the webservice returns the response

2XX: Successful responses

200: OK

201: Created

4XX: Client errors

404: Resource not found, wrong clients

405: Method not allowed

401: Unauthorized

5XX: Server errors

500: Internal Server error, for exceptions

By default the webservices itself is creating the status codes like 200, 500 and so on, but you can customize them using the ResponseEntity object

ResponseEntity: it is used to create a response with different status, content and so on.

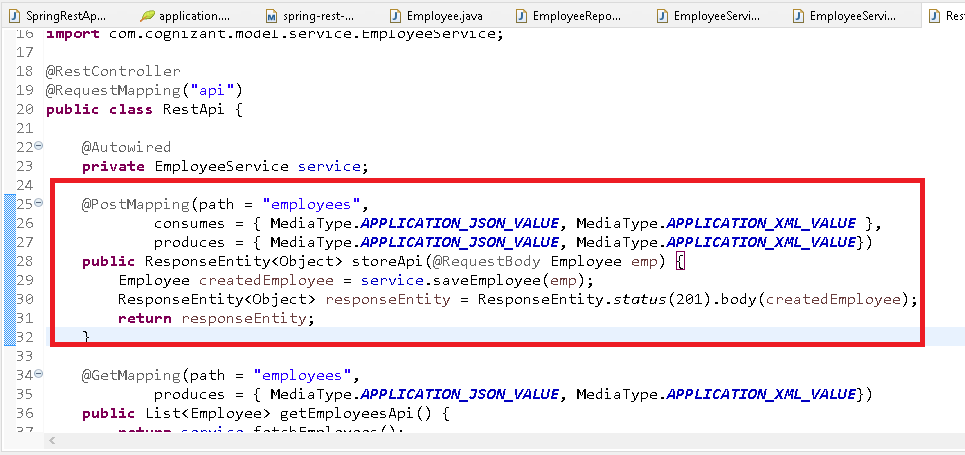
ResponseEntity is created using the static methods present in the ResponseEntity

Note: In real time you must return ResponseEntity instead of any java object

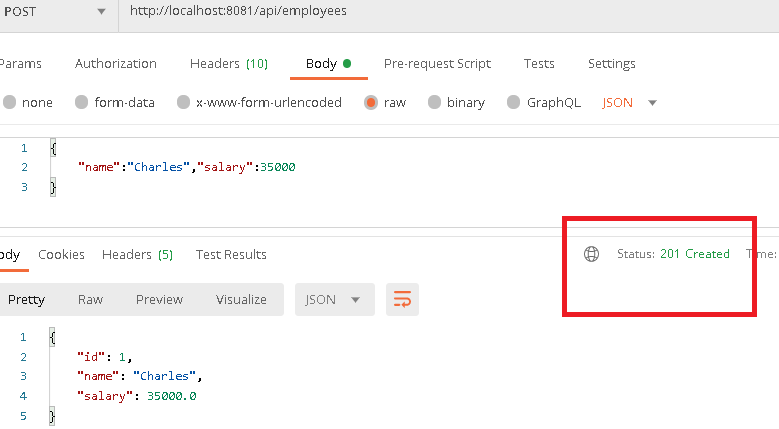
Ex: Using response entity you can return the content with any type  
@GetMapping  
public ResponseEntity<Object> greetings() { }

Ex: Using a java type will restrict only to return the data compatible to the return type  
@GetMapping  
public String greetings() { }

RestApi.java



Output:



Creating exception when data not found

We need to modify the service layer method to throw customized exception

EmployeeNotFoundException.java

**package** com.cognizant.exceptions;

@SuppressWarnings("serial")

**public** **class** EmployeeNotFoundException **extends** Exception {

**public** EmployeeNotFoundException() {

**super**("Sorry employee not found");

}

**public** EmployeeNotFoundException(String err) {

**super**(err);

}

}

Modify the service interface to use throws EmployeeNotFoundException

EmployeeService.java

**package** com.cognizant.model.service;

**import** java.util.List;

**import** com.cognizant.exceptions.EmployeeNotFoundException;

**import** com.cognizant.model.entities.Employee;

**public** **interface** EmployeeService {

**public** List<Employee> fetchEmployees();

**public** Employee saveEmployee(Employee employee);

**public** Employee fetchEmployee(**int** id) **throws** EmployeeNotFoundException;

}

Throw the exceptions if employee record is not available

EmployeeServiceImpl.java

**package** com.cognizant.model.service;

**import** java.util.List;

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

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

**import** com.cognizant.exceptions.EmployeeNotFoundException;

**import** com.cognizant.model.dao.EmployeeRepository;

**import** com.cognizant.model.entities.Employee;

@Service

**public** **class** EmployeeServiceImpl **implements** EmployeeService {

@Autowired

**private** EmployeeRepository employeeDao;

@Override

**public** List<Employee> fetchEmployees() {

**return** employeeDao.findAll();

}

@Override

**public** Employee saveEmployee(Employee employee) {

Employee created = employeeDao.save(employee);

**return** created;

}

@Override

**public** Employee fetchEmployee(**int** id) **throws** EmployeeNotFoundException {

**if**(employeeDao.findById(id).isPresent()) {

Employee employee = employeeDao.findById(id).get();

**return** employee;

}

**throw** **new** EmployeeNotFoundException("Sorry employee with an id "+id+" not found!");

}

}

Handling the exceptions in the RestApi

**package** com.cognizant.controller;

**import** java.util.List;

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

**import** org.springframework.http.MediaType;

**import** org.springframework.http.ResponseEntity;

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

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

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

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

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

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

**import** com.cognizant.exceptions.EmployeeNotFoundException;

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.service.EmployeeService;

@RestController

@RequestMapping("api")

**public** **class** RestApi {

@Autowired

**private** EmployeeService service;

@PostMapping(path = "employees",

consumes = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE*** },

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** ResponseEntity<Object> storeApi(@RequestBody Employee emp) {

Employee createdEmployee = service.saveEmployee(emp);

ResponseEntity<Object> responseEntity = ResponseEntity.*status*(201).body(createdEmployee);

**return** responseEntity;

}

@GetMapping(path = "employees",

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** List<Employee> getEmployeesApi() {

**return** service.fetchEmployees();

}

@GetMapping(path = "employees/{id}",

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** ResponseEntity<Object> getEmployeesByIdApi(@PathVariable **int** id) {

ResponseEntity<Object> responseEntity = **null**;

**try** {

Employee employee = service.fetchEmployee(id);

responseEntity = ResponseEntity.*status*(200).body(employee);

} **catch**(EmployeeNotFoundException exception) {

String message = exception.getMessage(); // {"message":"Sorry employee with id 100 not found"}

ResponseModel err = **new** ResponseModel(message);

responseEntity = ResponseEntity.*status*(404).body(err);

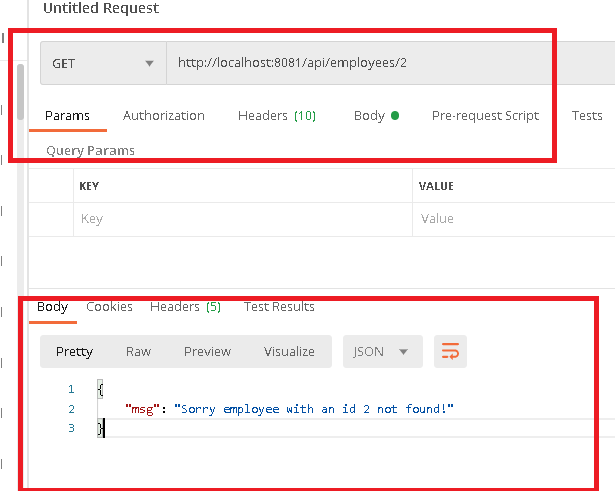
}

**return** responseEntity;

}

}

Output:



Ways to test the webservices

1. Postman application
2. Curl commands - You must have git-bash installed

@PutMapping & @DeleteMapping:

These methods are used when you want to perform update or delete operations

We will try to update the salary based on id and also we will try to delete the data based on id.

Note: In JpaRepository you will use save() method for both store & update, save() updates if the id exists else it stores the new record with new id

EmployeeService.java

**package** com.cognizant.model.service;

**import** java.util.List;

**import** com.cognizant.exceptions.EmployeeNotFoundException;

**import** com.cognizant.model.entities.Employee;

**public** **interface** EmployeeService {

**public** List<Employee> fetchEmployees();

**public** Employee saveEmployee(Employee employee);

**public** Employee fetchEmployee(**int** id) **throws** EmployeeNotFoundException;

**public** Employee updateSalary(**int** id, **double** salary) **throws** EmployeeNotFoundException;

**public** **void** deleteEmployee(**int** id)**throws** EmployeeNotFoundException;

}

EmployeeServiceImpl.java

**package** com.cognizant.model.service;

**import** java.util.List;

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

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

**import** org.springframework.transaction.annotation.Transactional;

**import** com.cognizant.exceptions.EmployeeNotFoundException;

**import** com.cognizant.model.dao.EmployeeRepository;

**import** com.cognizant.model.entities.Employee;

@Service

**public** **class** EmployeeServiceImpl **implements** EmployeeService {

@Autowired

**private** EmployeeRepository employeeDao;

@Override

**public** List<Employee> fetchEmployees() {

**return** employeeDao.findAll();

}

@Override

**public** Employee saveEmployee(Employee employee) {

Employee created = employeeDao.save(employee);

**return** created;

}

@Override

**public** Employee fetchEmployee(**int** id) **throws** EmployeeNotFoundException {

**if**(employeeDao.findById(id).isPresent()) {

Employee employee = employeeDao.findById(id).get();

**return** employee;

}

**throw** **new** EmployeeNotFoundException("Sorry employee with an id "+id+" not found!");

}

@Override

@Transactional

**public** Employee updateSalary(**int** id, **double** salary) **throws** EmployeeNotFoundException {

Employee emp = fetchEmployee(id);

emp.setSalary(salary);

employeeDao.save(emp);

**return** emp;

}

@Override

@Transactional

**public** **void** deleteEmployee(**int** id) **throws** EmployeeNotFoundException {

Employee emp = fetchEmployee(id);

employeeDao.delete(emp);

}

}

Create webservice to call update & delete

RestApi.java

**package** com.cognizant.controller;

**import** java.util.List;

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

**import** org.springframework.http.MediaType;

**import** org.springframework.http.ResponseEntity;

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

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

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

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

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

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

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

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

**import** com.cognizant.exceptions.EmployeeNotFoundException;

**import** com.cognizant.model.entities.Employee;

**import** com.cognizant.model.service.EmployeeService;

@RestController

@RequestMapping("api")

**public** **class** RestApi {

@Autowired

**private** EmployeeService service;

@PostMapping(path = "employees",

consumes = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE*** },

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** ResponseEntity<Object> storeApi(@RequestBody Employee emp) {

Employee createdEmployee = service.saveEmployee(emp);

ResponseEntity<Object> responseEntity = ResponseEntity.*status*(201).body(createdEmployee);

**return** responseEntity;

}

@GetMapping(path = "employees",

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** List<Employee> getEmployeesApi() {

**return** service.fetchEmployees();

}

@GetMapping(path = "employees/{id}",

produces = { MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** ResponseEntity<Object> getEmployeesByIdApi(@PathVariable **int** id) {

ResponseEntity<Object> responseEntity = **null**;

**try** {

Employee employee = service.fetchEmployee(id);

responseEntity = ResponseEntity.*status*(200).body(employee);

} **catch**(EmployeeNotFoundException exception) {

String message = exception.getMessage(); // {"msg":"Sorry employee with id 100 not found"}

ResponseModel err = **new** ResponseModel(message);

responseEntity = ResponseEntity.*status*(404).body(err);

}

**return** responseEntity;

}

@DeleteMapping(path = "employees/{id}",

produces = {MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**public** ResponseEntity<Object> deleteEmployeeByIdApi(@PathVariable **int** id) {

ResponseEntity<Object> responseEntity = **null**;

**try** {

service.deleteEmployee(id);

ResponseModel success = **new** ResponseModel("Employee with an id "+id+" deleted!");

responseEntity = ResponseEntity.*status*(200).body(success);

} **catch**(EmployeeNotFoundException exception) {

String message = exception.getMessage(); // {"msg":"Sorry employee with id 100 not found"}

ResponseModel err = **new** ResponseModel(message);

responseEntity = ResponseEntity.*status*(404).body(err);

}

**return** responseEntity;

}

@PutMapping(path = "employees/{id}/{salary}",

produces = {MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***} )

**public** ResponseEntity<Object> updateEmployeeByIdApi(@PathVariable **int** id, @PathVariable **double** salary) {

ResponseEntity<Object> responseEntity = **null**;

**try** {

Employee employee = service.updateSalary(id, salary);

responseEntity = ResponseEntity.*status*(200).body(employee);

} **catch**(EmployeeNotFoundException exception) {

String message = exception.getMessage(); // {"msg":"Sorry employee with id 100 not found"}

ResponseModel err = **new** ResponseModel(message);

responseEntity = ResponseEntity.*status*(404).body(err);

}

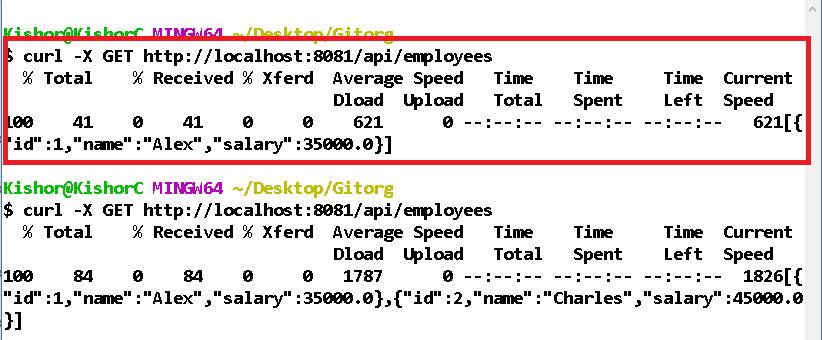
**return** responseEntity;

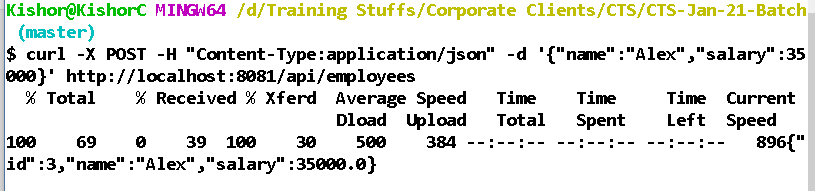
}

}

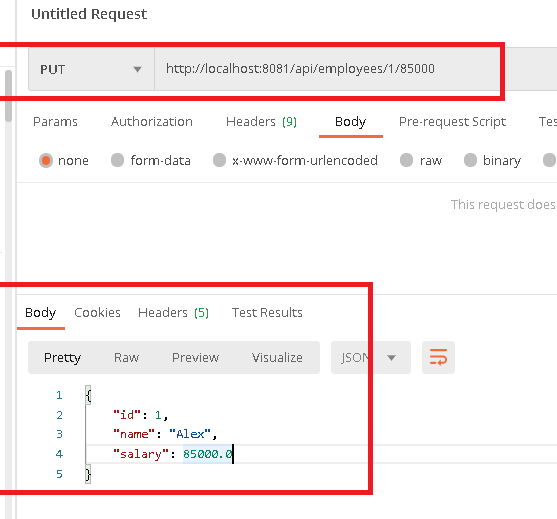
Testing the webservices using CURL

>> curl -X GET|POST|PUT|DELETE -d {} url

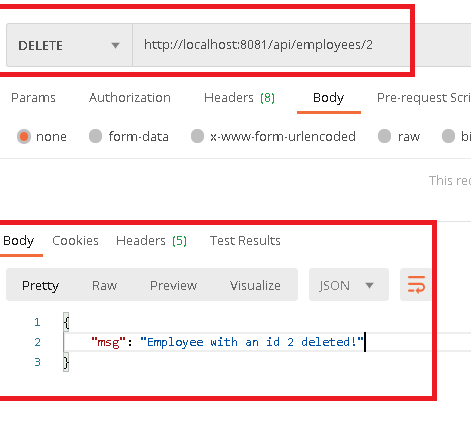




Performing update & delete



For delete operation



To integrate spring boot with angular you need angular application to use an instance of HttpClient

Note: Webservices don’t allow cross domain’s to exchange the data by default, hence you must enable webservice to allow different domain’s to exchange the data, else the client application gets CORS policy errors

How to enable different domain’s to exchange the data in spring boot

@CorsFilter({“ip”, “ip”}): list of machines that can exchange the data

@CorsFilter(“\*”): this allows all the machines to exchange the data

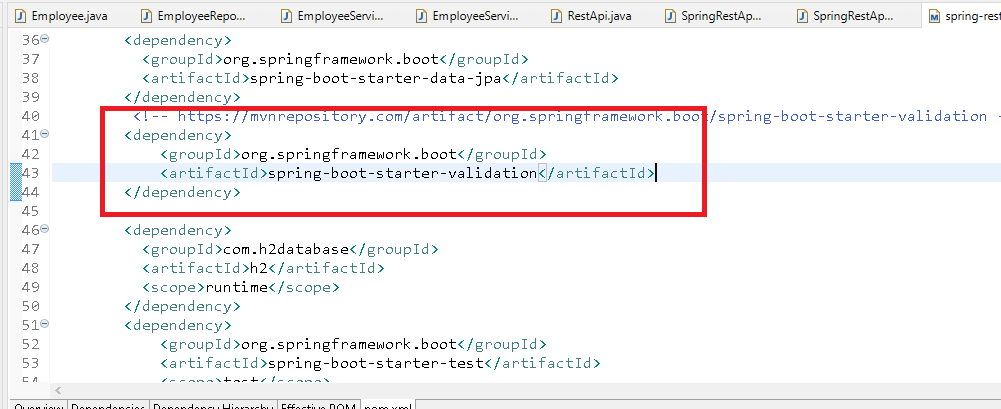
This annotation we must mention on top of the webservice class.

Bean Validations:

You can validate the bean properties against some inputs like @NotEmpty, @Size, @Min, @Max, @Positive, @Negative

These annotations you can use on top of the bean properties so that any input which fails against the validation you get some exceptions.

You need to use spring-boot-starter-validation library to use the annotations.



Now you can use annotations like @NotEmpty, @Min, @Max, @Positive, @Email and so on

All these annotations must imported from javax.validation.constraint.\*

Employee.java

**package** com.cognizant.model.entities;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** javax.persistence.Table;

**import** javax.validation.constraints.NotEmpty;

**import** javax.validation.constraints.Positive;

**import** javax.xml.bind.annotation.XmlRootElement;

**import** lombok.Getter;

**import** lombok.Setter;

**import** lombok.ToString;

@XmlRootElement

@Entity

@Table(name = "employee")

@ToString

@Getter

@Setter

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** **int** id;

@NotEmpty(message = "Name cannot be empty")

**private** String name;

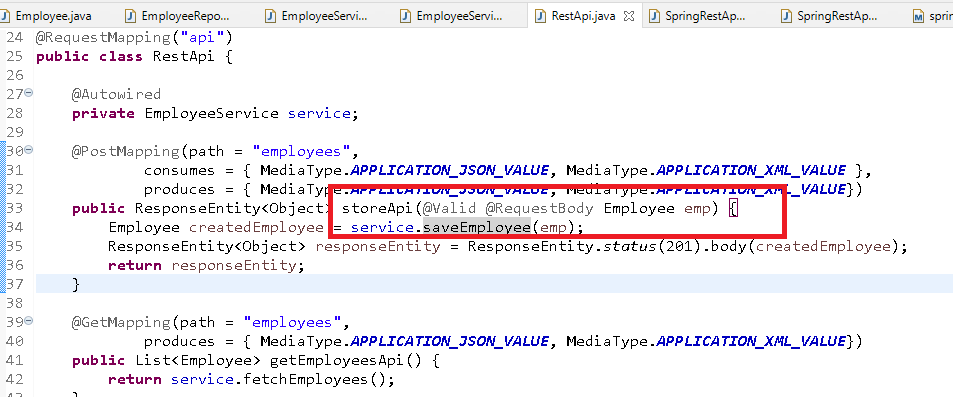
@Positive(message = "Salary cannot be negative")

**private** **double** salary;

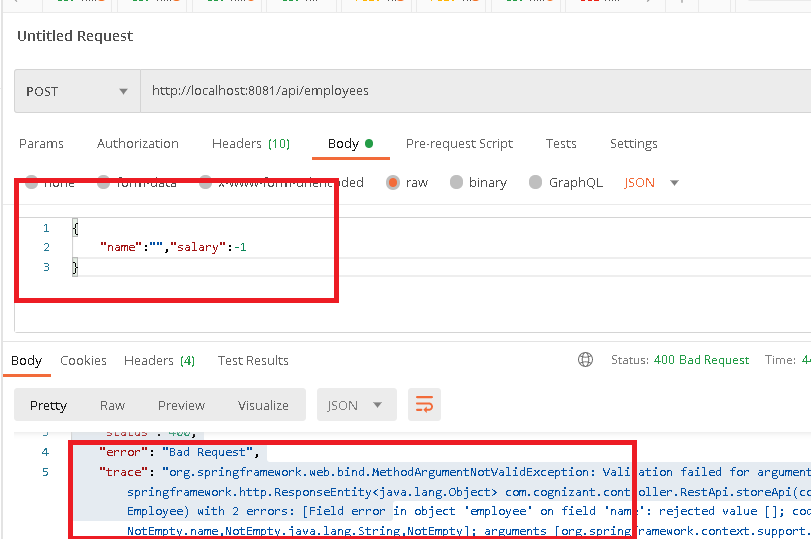
}

Now the request data must be validated hence you must use @Valid annotation in the method parameter which is taking the input

RestApi.java



The above will generate an exception if the input is not valid



For empty name & negative salary we get MethodArgumentNotValidException, but we may not be interested in the response instead we can handle this exception and customize the messages so that end user will get only error messages.

There are two ways

1. You can handle the exception can customize by creating your own exception, but this is specific to the method
2. You can create a global exception handler so that it can handle these exceptions across the applications

We can create a global exception handler so that it can handle the exceptions that occur anywhere

How to create Global Exception Handler

You must create a class with @ControllerAdvice which will intercept if any exceptions related to MethodArgumentNotValidException occurs, so it can handle the exceptions across the application.

Note: A class must extend one of the predefined exception class i.e., ResponseEntityExceptionHandler

GlobalCustomizedResponseException.java

**package** com.cognizant.exceptions;

**import** java.util.List;

**import** java.util.stream.Collectors;

**import** org.springframework.http.HttpHeaders;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.MethodArgumentNotValidException;

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

**import** org.springframework.web.context.request.WebRequest;

**import** org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

@ControllerAdvice

**public** **class** GlobalCustomizedResponseException **extends** ResponseEntityExceptionHandler {

@Override

**protected** ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,

HttpHeaders headers, HttpStatus status, WebRequest request) {

// this takes all the fields errors and adds the default messages to the list

List<String> errors = ex.getBindingResult()

.getFieldErrors()

.stream()

.map(err -> err.getDefaultMessage()).collect(Collectors.*toList*());

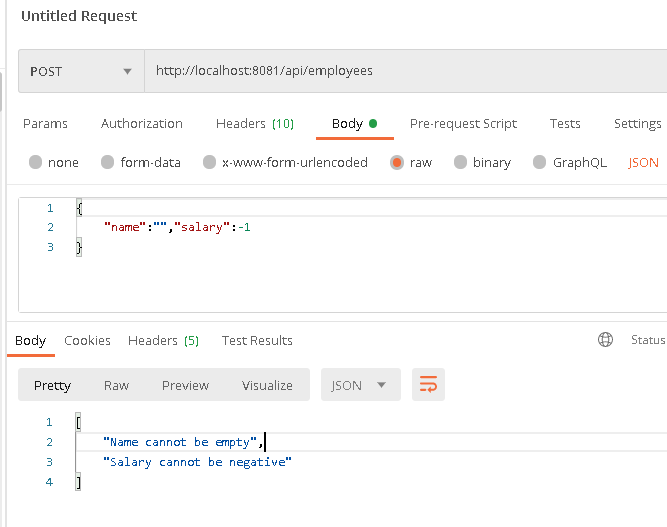
ResponseEntity<Object> response = ResponseEntity.*status*(404).body(errors);

**return** response;

}

}

Output:



Integrating Angular applications with Spring REST webservices

>> ng new employee-ui

Things to add in angular application

1. HttpClient: to call the webservices - HttpClientModule in the app.module.ts
2. FormBuilder: to send employee data to the webservices and you need ReactiveFormsModule
3. Service: to inject HttpClient
4. Components: to show the employee form & details.

Note: The restful webservice will not allow requests from another application by default hence you must use @CrossOrigin annotation on top of the class having webservices

>> ng g c employee-store  
>> ng g c employee-details  
>> ng g s employee

app.module.ts

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

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

import { AppRoutingModule } from './app-routing.module';

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

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

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

import { EmployeeDetailsComponent } from './employee-details/employee-details.component';

import { EmployeeStoreComponent } from './employee-store/employee-store.component';

@NgModule({

  declarations: [

    AppComponent,

    EmployeeDetailsComponent,

    EmployeeStoreComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    ReactiveFormsModule,

    HttpClientModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Adding employee-store & employee-details in the root component

app.component.html

<div>

  <h1>Employee UI to call spring rest services</h1>

  <hr />

  <app-employee-store></app-employee-store>

  <hr />

  <app-employee-details></app-employee-details>

</div>

employee.service.ts

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

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EmployeeService {

  baseUri : string = "http://localhost:8081/api";

  constructor(private httpClient : HttpClient) { }

  storeEmployee(formValues : any) : Observable<any> {

    let storeUri = `${this.baseUri}/employees`;

    return this.httpClient.post(storeUri, formValues);

  }

  getEmployee(id : number) : Observable<any> {

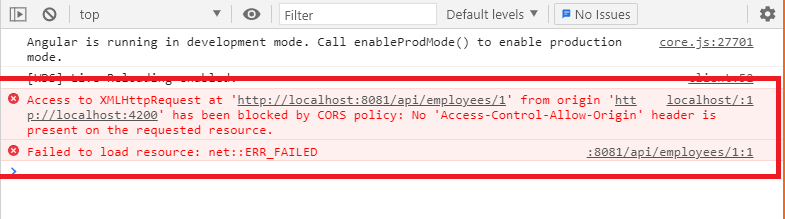
    let findUri = `${this.baseUri}/employees/${id}`;

    return this.httpClient.get(findUri);

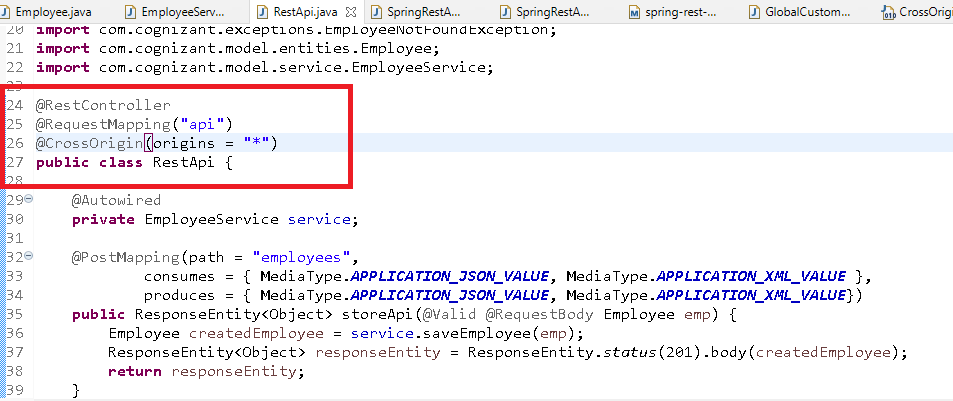
  }

}

When you send request to the spring you will get below error



You can notice localhost:8080/api is restricting localhost:4200, hence in spring @RestController you have to keep @CrossOrigin to allow the request from the different domain.



employee.service.ts

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

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

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class EmployeeService {

  baseUri : string = "http://localhost:8081/api";

  constructor(private httpClient : HttpClient) { }

  storeEmployee(formValues : any) : Observable<any> {

    let storeUri = `${this.baseUri}/employees`;

    return this.httpClient.post(storeUri, formValues);

  }

  getEmployee(id : number) : Observable<any> {

    let findUri = `${this.baseUri}/employees/${id}`;

    return this.httpClient.get(findUri);

  }

}

employee-details.component.ts

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

import { EmployeeService } from '../employee.service';

@Component({

  selector: 'app-employee-details',

  templateUrl: './employee-details.component.html',

  styleUrls: ['./employee-details.component.css']

})

export class EmployeeDetailsComponent implements OnInit {

  employee : undefined | any = undefined;

  message : undefined | string = undefined;

  constructor(private service : EmployeeService) { }

  ngOnInit(): void {

  }

  searchEmployee(id : number): void {

    this.service.getEmployee(id)

    .subscribe((success) => {

      this.employee = success;

      this.message = undefined;

    }, (err) => {

      this.message = err.error.msg;

      this.employee = undefined;

    });

  }

}

employee-details.component.html

<div>

    <h3>Search Employee by id</h3>

    Enter id<input type = "number" #tf >

    <input type = "button" value = "Search"

    (click) = "searchEmployee(tf.value)">

    <div \*ngIf = "employee">

        {{employee | json}}

    </div>

    <div \*ngIf = "message">

        {{message}}

    </div>

</div>

employee-store.component.ts

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

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

import { EmployeeService } from '../employee.service';

@Component({

  selector: 'app-employee-store',

  templateUrl: './employee-store.component.html',

  styleUrls: ['./employee-store.component.css']

})

export class EmployeeStoreComponent implements OnInit {

  constructor(

    private form : FormBuilder,

    private service : EmployeeService

    ) { }

  employee : any | undefined = undefined;

  employeeForm = this.form.group({

    name : [], salary : []

  });

  ngOnInit(): void {

  }

  handleSubmit() : void{

    this.service.storeEmployee(this.employeeForm.value)

    .subscribe(success => this.employee = success);

  }

}

employee-store.component.html

<div>

    <h3>Employee Form</h3><hr />

    <form [formGroup] = "employeeForm" (ngSubmit) = "handleSubmit()">

       Name: <input type = "text" formControlName = "name">

       <br />

       Salary : <input type = "text" formControlName = "salary">

       <br />

       <input type = "submit" value = "Store">

    </form>

    <div \*ngIf = "employee">

        {{employee | json}}

    </div>

</div>

Output:

