SPRING BOOT

- Create Spring Boot Project:
  - Using spring initializer: go to the link start.spring.io , and generate spring boot maven /gradle project for Intellij or eclipse
  - Directly create starter project in STS
- Project flow
  - Create spring starter project (maven)
    - Packaging as jar
  - Add dependencies
    - Spring boot starter web
  - Create controller, model, service, dao, exceptional handling
  - Build and package as jar
  - Run the jar

Spring Framework + Embedded HttpServers(Tomcat,Jetty) - @Configuration or XML<bean> Configuration = Spring Boot

- Reduces the configuration
- Reduces the number of dependencies and takes of dependent mgmt
- Standalone and production ready app
- Provide non functional features like(security, metrics, health checkups and externalized configuration)

Spring Boot is a Spring module which provides RAD (Rapid Application Development) feature to Spring framework.
It has extra support of auto-configuration and embedded application server like tomcat, jetty, etc.

In Spring framework, lot of focus is on the configuration and not in convention(coding). In order to avoid this, Spring Framework came into picture. Spring Boot is an extension to Spring. It gives all the dependencies, configuration and its main idea is to give production ready application.

SB provides jars for Embedded server(Tomcat), dependencies, configuration.
If we want to change any configuration manually, it can be done in application.properties file.

For standalone application: spring-boot-starter
For web application. : spring-boot-web-starter
For JDBC application : spring-boot-data-jdbc-starter
For JPA hibernate: spring-boot-data-jpa-starter

Features of Spring Boot:

———————————
Auto configuration
CommandLineRunner/ApplicationRunner. : During or before the launch of SB, if we want to initialize anything)
Externalised configuration
Support YAML
Auto redeploy with Devtools
Packaging as jar and war
Logging support (default one logbook), can change to log4j, jcl
Profiling

Creates stand-alone spring application with minimal configuration needed.
It has embedded tomcat, jetty which makes it just code and run the application.
Provide production-ready features such as metrics, health checks, and externalized configuration.
Absolutely no requirement for XML configuration.

Spring Boot key components:
——————————————-
Spring Boot auto-configuration.
Spring Boot starter POMs. (Project Object Model.)
Spring boot parent starter (to manage the child dependencies)
Spring Boot CLI.

Spring Boot Actuators.

What are starter dependencies?
————————————————-
Spring boot starter is a maven template that contains a collection of all the relevant transitive dependencies that are needed to start a particular functionality.
Like we need to import spring-boot-starter-web dependency for creating a web application.

```
<dependency>
<groupId> org.springframework.boot</groupId>
<artifactId> spring-boot-starter-web </artifactId>
</dependency>
```

Spring boot Starter dependencies:
————————————————

Data JPA starter.
Test Starter.

Security starter.
Web starter.
Mail starter.
Thymeleaf starter.
Here Thymeleaf(default HTML template engine in SB)

How does Spring Boot works?
——————————————-
Spring Boot automatically configures your application based on the dependencies you have added to the project by using annotation. The entry point of the spring boot application is the class that contains @SpringBootApplication annotation and the main method.
Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.
What is Spring Initializer?
Spring Initializer is a web application that helps you to create an initial spring boot project structure and provides a maven or gradle file to build your code. It solves the problem of setting up a framework when you are starting a project from scratch.
What is Spring Boot CLI and what are its benefits?
Spring Boot CLI is a command-line interface that allows you to create a spring-based java application using Groovy.
Example: You don't need to create getter and setter method or access modifier, return statement. If you use the JDBC template, it automatically loads for you.
What are the most common Spring Boot CLI commands?
-run, -test, -grap, -jar, -war, -install, -uninstall, --init, -shell, -help.

Advanced Spring Boot Questions
What Are the Basic Annotations that Spring Boot Offers?
The primary annotations that Spring Boot offers reside in its org.springframework.boot.autoconfigure and its sub-packages. Here are a couple of basic ones:
@EnableAutoConfiguration – to make Spring Boot look for auto-configuration beans on its classpath and automatically apply them.
@SpringBootApplication – used to denote the main class of a Boot Application. This annotation combines @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.
What is Spring Boot dependency management?
Spring Boot dependency management is used to manage dependencies and configuration automatically without you specifying the version for any of that dependencies.

Can we create a non-web application in Spring Boot?
Yes, we can create a non-web application by removing the web dependencies from the classpath along with changing the way Spring Boot creates the application context.
Is it possible to change the port of the embedded Tomcat server in Spring Boot?
Yes, it is possible. By using the server.port in the application.properties.
server.port =
What is the default port of tomcat in spring boot?
The default port of the tomcat server-id 8080. It can be changed by adding sever.port properties in the application.property file.
Can we override or replace the Embedded tomcat server in Spring Boot?
Yes, we can replace the Embedded Tomcat server with any server by using the Starter dependency in the pom.xml file. Like you can use spring-boot-starter-jetty as a dependency for using a jetty server in your project.
Can we disable the default web server in the Spring boot application?
Yes, we can use application.properties to configure the web application type i.e spring.main.web-application-type=none.
How to disable a specific auto-configuration class?
You can use exclude attribute of @EnableAutoConfiguration if you want auto-configuration not to apply to any specific class.
//use of exclude
@EnableAutoConfiguration(exclude={className})
Explain @RestController annotation in Sprint boot?
It is a combination of @Controller and @ResponseBody, used for creating a restful controller. It converts the response to JSON or XML. It ensures that data returned by each method will be written straight into the response body instead of returning a template in the browser.
What is the difference between @RestController and @Controller in Spring Boot?
@Controller Map of the model object to view or template and make it human readable but @RestController simply returns the object and object data is directly written in HTTP response as JSON or XML.

Describe the flow of HTTPS requests through the Spring Boot application?
On a high-level spring boot application follow the MVC pattern which is depicted in the below flow diagram.

What is the difference between RequestMapping and GetMapping?
RequestMapping can be used with GET, POST, PUT, and many other request methods using the method attribute on the annotation. Whereas getMapping is only an extension of RequestMapping which helps you to improve on clarity on request.
What is the use of Profiles in spring boot?

While developing the application we deal with multiple environments such as dev, QA, Prod, and each environment requires a different configuration. For eg., we might be using an embedded H2 database for dev but for prod, we might have proprietary Oracle or DB2. Even if DBMS is the same across the environment, the URLs will be different.
To make this easy and clean, Spring has the provision of Profiles to keep the separate configuration of environments.
What is Spring Actuator? What are its advantages?
An actuator is an additional feature of Spring that helps you to monitor and manage your application when you push it to production. These actuators include auditing, health, CPU usage, HTTP hits, and metric gathering, and many more that are automatically applied to your application.
How to enable Actuator in Spring boot application?
To enable the spring actuator feature, we need to add the dependency of “spring-boot-starter-actuator” in pom.xml.

```
<dependency>
<groupId> org.springframework.boot</groupId>
<artifactId> spring-boot-starter-actuator </artifactId>
</dependency>
```

What are the actuator-provided endpoints used for monitoring the Spring boot application?
Actuators provide below pre-defined endpoints to monitor our application -
Health
Info
Beans
Mappings
Configprops
Httptrace
Heapdump
Threaddump
Shutdown

How to get the list of all the beans in your Spring boot application?
Spring Boot actuator “/Beans” is used to get the list of all the spring beans in your application.
How to check the environment properties in your Spring boot application?
Spring Boot actuator “/env” returns the list of all the environment properties of running the spring boot application.
How to enable debugging log in the spring boot application?
Debugging logs can be enabled in three ways -
We can start the application with --debug switch.
We can set the logging.level.root=debug property in application.property file.
We can set the logging level of the root logger to debug in the supplied logging

configuration file.

Where do we define properties in the Spring Boot application?
You can define both application and Spring boot-related properties into a file called application.properties. You can create this file manually or use Spring Initializer to create this file. You don't need to do any special configuration to instruct Spring Boot to load this file, If it exists in classpath then spring boot automatically loads it and configure itself and the application code accordingly.

What is an IOC container?
IoC Container is a framework for implementing automatic dependency injection. It manages object creation and its life-time and also injects dependencies into the class.

-- Dependency Injection:
The process of injecting dependent bean objects into target bean objects is called dependency injection.
Dependency injection is a technique in which an object receives other objects that it depends on, called dependencies.
Typically, the receiving object is called a client and the passed-in ('injected') object is called a service
Dependency Injection is a fundamental aspect of the Spring framework, through which
the Spring container "injects" objects into other objects or "dependencies".
This allows for loose coupling(objects don't depend on one another totally, they can be allowed for a change)
of components and moves the responsibility of managing components onto the container

Example: A Car class might need a reference to an Engine class. These required classes are called dependencies,
and in this example the Car class is dependent on having an instance of the Engine class to run.
-- Advantages of DI:
1. To achieve loose coupling(1 object should not totally dependent on the other).
Example:
Laptop {
HardDrive obj;
Ram ob;
}

We do abstraction (by creating an abstract class or an interface on top of this HardDrive to incorporate any
changes(multiple hard drive companies like Hitachi, Samsung)

HardDrive obj = new Hitachi(); or samsung();
Here , we shunt hardcode, so the hard drive obj is injected into Laptop.
This injection is done by the external containers in Spring framework.
Dependency Injection below:

```
Class Laptop {
@autowired
HardDrive obj;
}

@Component
Class Hitachi implements HardDrive {
}
```

2. For easy Testing: Only if our app is loosely coupled testing can be done on the objects without affecting each other.
For example, a mock obj can be created on the database and tested with the app obj without affecting the database.

3. Object does not look up for dependencies.
4. Cleaner code
5. Externalization (no need recompilation for system changes)
6. Classes and modules are independent

-- There are 3 ways to perform DI:
1. CONSTRUCTOR INJECTION:
  Dependency injection is done through constructor]
  This type of injection is safer as the objects won't get created if the dependencies aren't available
  It helps in creating immutable objects because a constructor's signature is
  the only possible way to create objects. Once we create a bean, we cannot alter its dependencies anymore.
Also when there are multiple objects to be autowired, contructor injection can be used.

Example Program:

```
@Component
public class Alien2 {
	Laptop laptop;
	@Autowired
	Alien2(Laptop laptop) {
		System.out.print("This is dependency injection through constructor" );
		this.laptop =laptop;
	}
	public void show() {
		System.out.println("In show method - Alien 2");
		laptop.compile();
	}
}
```

2. SETTER INJECTION
 Setter injection is a dependency injection in which the spring framework injects
 the dependency object using the setter method.
 Here in the setter method injection, the object is created first and then the
dependency is injected.

Example Program:

```
@Component
public class Alien3 {
	Laptop laptop;
	@Autowired
	void setLaptop(Laptop laptop) {
		System.out.print("This is dependency injection through setter
method" );
		this.laptop = laptop;
	}
	Laptop getLaptop() {
		return laptop;
	}
	public void show() {
		System.out.println("In show method - Alien 3");
		laptop.compile();
	}
}
```

3. FIELD LEVEL OR CLASS LEVEL INJECTION

-- @Autowired
Autowiring feature of spring framework **enables you to inject the object dependency implicitly**. It internally uses setter or constructor injection.
Autowiring can't be used to inject primitive and string values. It works with reference only.
Autowiring is a great technique used to reduce the wiring up and configuration of code.
Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection.
To autowire our applications using the Java configuration, we just simply need to add a @Autowired to our constructor, field, setter method, or config method.

It automatically searches for the object and by default autowire searches for the TYPE not by name of the object.
The default name of the class object in the container is the lowercase of the class name.
Example : For the class Laptop, the object name is laptop , the object type is Laptop.class
We can overwrite the class name giving different name at @component (“lap1”)

Note: Default search : class TYPE (Laptop.class)
@Qualifier : the autowire searches for the object by name (laptop),

-- SPRING SCOPES
There are 5 scopes available. Requires AOP jar for @scope annotation

-SINGLETON - It is a default scope. Only one instantiation. Single instance per Spring container.
RealTime Usage: You can store state unique to that application in there, but you don't want to store state unique to that user inside that object.
- PROTOTYPE - It is a new bean, a unique instance per request.
Note: If we don’t want to see the object created at all, it is possible in priototype(don’t call get bean)

Valid only in web-awre spring projects
- REQUEST - It returns a bean per HTTP request, which sounds a lot like prototype except it's for the lifecycle of a bean request,
which is fairly short, but longer than prototype where it's one instance per every time I ask the container for a bean.
- SESSION - The session just returns a single bean per HTTP session, and that will live as long as that user session is alive,
so 10 minutes, 20 minutes, 30 minutes, however long they're alive on that website a bean of scope session will stick around.
- GLOBAL - It returns a single bean per application, so once I access it, it's alive for the duration of that application,
not just my visit to that application. You could think of it as singleton, but it's really the entire life of that
application on the server until it gets undeployed or the server gets rebooted.

Spring Boot application by extending SpringBootServletInitializer, it creates a dispatcher servlet and start serving up things.
And as part of that annotation of @SpringBootApplication, it goes and starts looking for the controllers(Greeting and Regirstration) and the annotations such as @Controller and @GetMapping.

— Externalization: create external file and have generic values to be passed for the app
- using @value
- using @configurationProperties

— Logging
Spring boot by default has logging feature spring-boot-starter-logging
If we want custom logging, use log4j, by adding spring-boot-starter-log4j2 dependency and with log4j.xml file for custom configuration

— Profiling
- To configure different environment databases (dev, uat, prod) , we define
    - application.properties
    - application.dev.properties
    - application.prod.properties
    - application.uat.properties
- IT can be configured using yaml file also

— Health status health check also can be done using. @Profile annotation
Can be passed in command line using
java -jar spring-boot-core-profile-1.0.jar --spring.profiles.active=dev

— compare diff web servers
https://www.baeldung.com/spring-boot-servlet-containers

SPRING BOOT WEB APP
————————————————

Webapp using Spring boot
Application properties file:
Spring boot provides auto configuration but we can even manually provide configuration if any path is changed from default directory structure.
Example:
Jsp pages or any other pages can be configured manually in application.preperties file
Prefix - path of the file
Suffix - extension of the file
Webapp using Spring Boot accepting client data
Webapp using Spring Boot ModelAndView

Dispatcher Servlet:
The DispatcherServlet is the front controller in Spring web applications. It's used to create web applications and REST services in Spring MVC.
In a traditional Spring web application, this servlet is defined in the web. ... xml file

to DispatcherServlet in a Spring Boot application.

ModelAndView:
Represents a model and view returned by a handler, to be resolved by a DispatcherServlet.
The view can take the form of a String view name which will need to be resolved by a ViewResolver object;
alternatively a View object can be specified directly. The model is a Map, allowing the use of multiple objects keyed by name.

viewResolver:
Spring uses ViewResolver to translate the view names in @Controller to actual View.
Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. ...
The two interfaces that are important to the way Spring handles views are ViewResolver and View . The ViewResolver provides a mapping between view names and actual views.

Webapp using Spring Boot Model object : Passing an object into ModelAndview key and getting the object values in the view.
Spring Boot JPA MVC H2 Examples
    Connecting to H2 embedded database, create table, insert data into it , insert data using spring boot, and querying the table using spring boot in model and view format and in json format
Postman , Data JPA, MVC, H2, REST Example
    Postman is a client side application.
    Postman is an application used for API testing.
    It is a HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we obtain different types of responses that need to be subsequently validated.

Note: Jackson core jar file present in Maven dependency is responsible to convert the Java object data into JSON format

Content Negotiation : It helps the Consumer and Provider interact about the data exchange format. (JSON or XML in this example)
    In the maven dependency, add Jackson format XML dependency
REST API:
    Post Request (POST): data sent in JSON format , saved into repo and returned in JSON format using Postman API. (Server consumes data)
    Get Request (GET): return data from repo in JSON format (Server produces data)
    Delete Request (DELETE)

Update or Save Request (PUT)

RESTful Service:
Communication of data occurs in JSON format.
CRUD, Get all students, Patch(updating the required field)

REST Standards:
POST /students
GET /students
GET /students/{student_id}
PUT /student/{student_id}
DELETE /student/{student_id}
PATCH /student/{student_id}

Request Pay load. : the json which I send to server through postman
Response Pay load: the json which we get from server

What is RESTful service?
HTTP Status codes
HttpSession Management

Spring Data REST. (NO CONTROLLER) services directly hit the repository
Spring Data REST is a framework that builds itself on top of the applications data repositories and expose those repositories in the form of REST endpoints.
In order to make it easier for the clients to discover the HTTP access points exposed by the repositories, Spring Data REST uses hypermedia driven endpoints.

What is JPA?
——————
The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database.
JPA is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java Industry.
JPA is specification, and requires an implementation.
JPA also defines a runtime EntityManager API for processing queries and transaction on the objects against the database.

ORM (Object Relational Mapping)
Java Objects with data are mapped to the database
Class name - Table name
Class variables - table columns

NOTE:
Hibernate , iBatis, TopLink are the tools to implement the ORM applications where the data is mapped between Java objects and the databases.
With the help of JPA, it is easy to switch between these tools if needed based on the requirements.
Because JPA is a specification where the app is built and the implementation can be any of the tools Hibernate or iBatis or TopLink.

— application.properties file

```
spring.h2.console.enabled=true
spring.datasource.platform=h2
spring.datasource.url=jdbc:h2:mem:harini
spring.jpa.defer-datasource-initialization=true

spring.datasource.url=jdbc:postgresql://localhost:5432/orbitz
spring.datasource.username=postgres
spring.datasource.password=root

# Show or not log for each sql query
spring.jpa.show-sql=true
# Hibernate ddl auto (create, create-drop, update): with "create-drop" the database
# schema will be automatically updated for every start of application
spring.jpa.hibernate.ddl-auto=update

# Allows Hibernate to generate SQL optimized for a particular DBMS
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

server.servlet.context-path=/rates

#logging.file.name=logfile.txt

management.endpoints.web.exposure.include=*

server.port=
```

To Study:
RESTful service theory parts learn the below conceptS
HTTP, HTTP Session management , HTTPS status codes, client, server, Request Payload, Response Payload, API
Microservices integration

Service to Service call (Examples Student & Department, Online shoppng(Shipment & Price services.)
Performance issues Handling, Error handling based on the HTTP status codes
Core Java, Servlets, JSP, HTTP , Java basic programs, data structure programs, spring boot, sprint mvc, spring core   notes.
Hibernate video
View resolver

— Spring Boot core App:

Here, no controller

```
@SpringBootApplication
class MySpringBootApp
{
```

```
public static void main(String args[])
{
ConfigurableApplicationContext context =
SpringApplication.run(MySpringBootApp.class, args);
Customer c =  context.getBean(Customer.class); // gives bean object , here
Customer bean object
c.show(); // methods in the Customer class
}
}

@Component.  // a class level annotation, makes a class as spring bean (which
gives objects automatically)
 class Customer {
}
```

— Spring Boot Web App. Without REST , covers Spring MVC without database

Here, controller, service, repository annotations are used

```
@SpringBootApplication
class MySpringBootWebApp
{
 public static void main(String args[])
{
SpringApplication.run(MySpringBootAWebApp.class, args);
}
}

@Controller
class HomeController
{

@RequestMapping("call")
Public ModelAndView customerView(Customer customer) {. // here customer
object details are passed as a
                // client request, the data is obtained in a view with customer details
ModelAndView mv = new ModelAndView();
mv.addObject("keyObj",customer);
mv.setViewName("home");
return mv;
}
/*
	@RequestMapping("call")
	public String home(@RequestParam("name")String myName, HttpSession 
session)
	 {	    // without req & res objects, we can get session object through DI by 
spring
	 // @RequestParam to extract query parameters, form parameters, and even 
files from the request.
	// here it maps name = myName
		System.out.println("Hi " + myName);
		session.setAttribute("attributeName", myName);
		return "home";
	}
*/

}
```

home.jsp :

```
<html> <head> </hrad><body>
Welcome ${keyObj.aid} , ${keyObj.aName}, ${keyObj.lang}
</body> </html>
```

Application.properties file: (also add japer dependency in pom.xml file)

```
spring.mvc.view.prefix=/pages/
spring.mvc.view.suffix=.jsp
```

Output:

```
  http://localhost:8080/call?aid=1&aName=harini&lang=telugu
```

In pom.xml:
To read jsp page

```
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.55</version>
</dependency>

<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
```

— Spring Boot Web App. Without REST with database

```
/*
Data seen on the browser in JSON format by default or in Model and View form
using jsp. It is not REST application
*/

@SpringBootApplication
class MySpringBootWebAppDB {
public static void main(String args[]) {
SpringApplication.run(MySpringBootWebAppDB.class, args);
}}

@Controller
Class CustomerController {

// add a customer
@RequestMapping(path=“/customer”)
Customer addCustomer(Customer customer) {
repo.save(customer);
return customer;
}


// get a customer
@RequestMapping(path=“/customer/{cusId}”)
@ResponseBody
Optional<Customer>  getCustomer(@PathVariable(“cusId”) int id) {
return repo.findById(id);
}

// get all customers
/*
@RequestMapping(path= “/customers”, produces= {"application/xml"})  //  to
restrict spring boot to support only XML format from server side. here produces
means server produces data to the client, if consumes, the server consumes data
from the client
*/

@RequestMapping(path=“/customers”)
@ResponseBody
List<Customer> getCustomers() {
  Return repo.findAll();
}

// delete a customer

@RequestMapping(path="/customer/id")
String deleteCustomer(@PathVariable("cusId") int id) {
repo.deleteById(id);
return "deleted";
}



customer.dao:
interface CustomerRepo extends JpaRepository<Customer,Integer> {
/*
	ArrayList<Customer> findByLang(String lang);
	ArrayList<Customer> findByAidGreaterThan(int aid);
	@Query("from Customer where lang=?1 order by a_name")
	ArrayList<Customer> findByLangSorted(String lang);
*/
}

Model:
Class Customer {
@Entity
public class Customer {
	@Id
	private int cusId;
	private String cusName;
	private String lang;
Getters & setters }
}
```

Refer at the bottom for the Database connection details in application.properties and pom.xml file

— Spring Boot Web App with REST with database & REST TEMPLATE

REST @RestController = @Controller + @ResponseBody
REST Representational State transfer
REST application produces and consumes JSON data or xml data (it accepts request and produces response in json format by default)
REST is not a protocol like HTTP. It is a set of guidelines to build a scalable,

performant,fault-tolerant and extendable system.
In order for an API to be RESTful, it has to adhere to 6 constraints:
Uniform interface :
Client — server separation
Stateless
Layered system
Cacheable
Code-on-demand

Rest Template:
It is used to call one microservice from another microservice.
MicroService: A monolithic application is split into small services which are independent, loosely coupled, easily maintainable.

Challenges with micro services:
Design. ...
Security. ...
Testing. ...
Increased operational complexity. ...
Communication. ...
Your defined domain is unclear/uncertain. ...
Improved efficiency isn't guaranteed. ...
Application size is small or uncomplex.

How do micro service work?
First, software functionality is broken down into small, isolated modules. These modules have defined, stand-alone tasks. Then, the system works by linking these small pieces of software together using application programming interfaces (APIs).

App:
```
@SpringBootApplication
public class SpringBootJpaApp {
public static void main(String args[])  {
SpringApplication.run(SpringBootJpaApp.class, args);
}}
```

```
@RestController
Class SpringBootJpaRest {

@Autowired
CustomerService customerService;


//get a customer
@GetMapping("/customer/{cusId}")
Customer getCustomer(@PathVariable("cusId) int id) {
   return customerService.getCustomer(id);

//get all customers
@GetMapping("/customers")
List<Customer> getCustomers() {
return customerService.getCustomers();
}

// add customer
@PostMapping("/customer")
Customer addCustomer(@RequestBody Customer customer)  // send the data
from the client
{
  customerService.addCustomer(customer);
}

// update customer
@PutMapping("/customer")
Customer saveOrUpdateCustomer(@RequestBody Customer customer)
 {
return customerService.updateCustomer(customer);
}
//delete a customer

@DeleteMapping("/customer/{cusId}")
void deleteCustomer(@PathVariable("cusId") int id) {
customerService.deleteCustomer(id);

}


@Service
```

```
Class CustomerService {


CustomerRepo repo;
RestTemplate addressRestTemplate;

@Autowired
CustomerService(RestTemplate addressRestTemplate, CustomerRepo repo) {
this.addressRestTemplate = addressRestTemplate;
this.repo = repo;

//get a customer with Address
Customer getCustomer(int id) {
Customer customer = repo.findById(id).orElse(null);

Address address = addressRestTemplate
        .getForEntity("http://localhost:8090/address" + customer.addressId,
Address.class).getBody();
customer.setAddress(address);
return customer;
}

// get all customers
List<Customer> getCustomers() {
return repo.findAll();
}

// add a customer
Customer addCustomer(Customer customer) {
repo.save(customer);
return Customer;
}

// delete customer
void deleteCustomer(int id) {
repo.deleteById(id);
}


// update Customer
Customer updateCustomer(Customer customer) {
repo.save(customer);
return customer;
}
}

@Configuration
public class CustomerConfig {
// Create Bean for the Rest Template to autowire the Rest Template Object

@Bean
public RestTemplate addressRestTemplate(RestTemplateBuilder
restTemplateBuilder) {
return restTemplateBuilder.build;
}
}
```

— Database connection in Spring Boot:

1., To connect to Postgres database
In application.properties file:

```
spring.dataasource.url=jdbc.postgresql://localhost:5432/database_name
Spring.datasource.username=postgres
spring.datasource.password=root
```

In pom.xml:

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>

		<dependency>
			<groupId>org.postgresql</groupId>
			<artifactId>postgresql</artifactId>
			<scope>runtime</scope>
		</dependency>
```

2. To connect to in memory h2 database
In application.properties file:

```
spring.h2.console.enabled=true
spring.datasource.platform=h2
spring.datasource.url=jdbc:h2:mem:harini
spring.jpa.defer-datasource-initialization=true
```

In pom.xml:

```
      <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-data-jpa</artifactId>
       </dependency>


       <dependency>
           <groupId>com.h2database</groupId>
           <artifactId>h2</artifactId>
           <scope>runtime</scope>
       </dependency>
```