

Microservices with Spring Boot and Spring Cloud

Spring Boot:

Spring boot is extension of Spring framework with the extensive support for Microservice architecture.

Why Spring Boot for Microservice Applications or any Applications:

1.Easy Dependency Management

(dependencies starters)

2.Auto Configuration:

Applications. Properties/yml

3.Embedded Servers.

Default spring boot will support Tomcat/Jetty/Undertwo.

Spring V/S Spring Boot:

Spring	Spring Boot
spring don't have any embedded servers. Spring is JEE framework to build WebApplciaitons. Primary feature is Dependency Injection by Using IOC.	Spring Boot had its own embeded containers like tomcat,jetty,Undertwo. Spring Boot a frame work wiedy used to develop rest API.
Spring Application its just 3 layered architecture view ,server,db	Primary feature is Auto Configuration It automatically configures classes based on requirement.
Spring you need collect all dependencies manually	Spring will give you rest based architecture i.e Distributed Applciations. Spring Boot comes with starter in pom.xml

Mavan V/S Gradle

Maven	Gradle
Build descriptor is POM.xml	Build descriptor is build.gradle
Strictly uses JAVA	Internally uses groovy engine so that run time dependency change possible.
Dependencies are mention in the form of <tags>	Dependencies are mention in the form of JSON

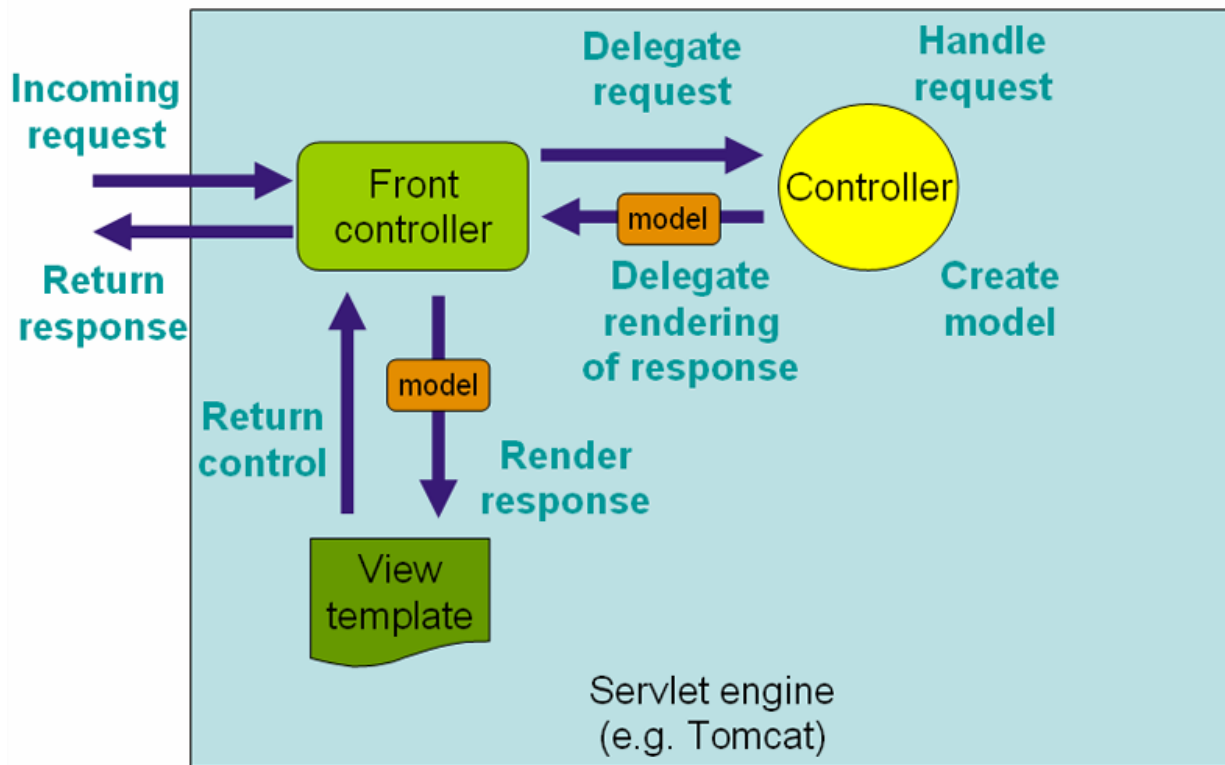
@SpringBootApplication=Autoconfiguration+Componentscan.

SpringApplication.run() will initialize IOC in SpringBoot Application.

Spring MVC:

Spring's web MVC framework is designed around a central Servlet (**DispatcherServlet**) that dispatches requests to controllers and offers other functionality that facilitates the development of web applications

DispatcherServlet is an expression of the “**Front Controller**” design pattern.



In spring boot to achieve the interoperability instead of `@Controller` will use `@RestController`.

`@RestController = @Controller + @ResponseBody`.

Ex:

```
@RestController
public class DemoController {

    @RequestMapping(path = "/demo", produces = "text/text")
    public String greet() {
        return "Welcome to demo";
    }
}
```

Inversion of Controller (IOC): IOC is design pattern which delegates the control of object creation to avoid the tight coupling.

IOC will be implemented by Dependency Injection.

Mainly 3 types of IOC containers:

1.BeanFactory (Spring 2 default)

2.ApplicationContext (Spring 3 onwards)

3.ConfigurableApplicationContext (Spring Boot default).

IOC container is a entity which perform IOC mechanism.

Dependency Injection(DI):

Hand overring the appropriate dependencies to the corresponding dependents called as DI.

Dependency: An object usually requires for the other objects to perform their operation. we can call this objects are dependencies.

Injection: the process of providing the required dependencies injection.

Dependency Injection are categorized into 3 types:

1.Constructor Injection:

In constructor Injection the dependencies required for the class are provided as arguments to the constructor.

```
@Component
public class Sandwich {
    Catagory catagory;
    @Autowired
    public Sandwich(Catagory catagory) {
        this.catagory = catagory;
    }
    public Catagory getCatagory() {
        return catagory;
    }
    public void setCatagory(Catagory catagory) {
        this.catagory = catagory;
    }
}
```

Sandwich.java

```
@Component
public class Catagory {

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

catagory.java

Field Level Injection:

Field level injection spring assign's the required dependencies directly to the fields on annotating with @Autowired.

```
@Component
public class Sandwich {
    @Autowired
    Catagory catagory;
    public Sandwich(Catagory catagory) {
        this.catagory = catagory;
    }

    public Catagory getCatagory() {
        return catagory;
    }

    public void setCatagory(Catagory catagory) {
        this.catagory = catagory;
    }
}
```

Sandwich.java

Setter Injection:

Setter Injection spring assigns the required dependencies directly to the setter methods on annotating with @Autowired

```
@Component
public class Sandwich {
    Catagory catagory;
    public Sandwich(Catagory catagory) {
        this.catagory = catagory;
    }
    public Catagory getCatagory() {
        return catagory;
    }

    @Autowired
    public void setCatagory(Catagory catagory) {
        this.catagory = catagory;
    }
}
```

Sandwich.java

Note:

For mandatory dependencies we will use Constructor Injection.

For optional dependencies we will use either setter injection on field injection.

PathVariable:

@PathVariable is a spring annotation Which indicates that method parameter should bound with to URI variable.

```
@RestController
public class UserController {

    @GetMapping("/user/{name}")
    public User getUser(@PathVariable String name) {

        User user1 = new User();
        user1.setId(1);
        user1.setName("vivek");
        user1.setEmail("vivek@amazoncom");

        User user2 = new User();
        user2.setId(2);
        user2.setName("anand");
        user2.setEmail("anand@amazon.com");

        List<User> userList = new ArrayList<User>();
        userList.add(user1);
        userList.add(user2);

        for (User user : userList) {
            if(user.getName().equalsIgnoreCase(name)) {
                return user;
            }
        }

        return null;
    }
}
```


Converting above for into lambda:

```
userList.stream().filter(temp->  
temp.getName().equals(name)).collect(Collectors.toList()).get(0);
```

@RequestParam:

@RequestParam is spring annotation used to bind a parameter to method through Web request.

We can consume the api with request param in below format:

<http://localhost:8080/user?user=vivek>

```
@GetMapping(path = "/user", produces = "text/html")  
public String getUser(@RequestParam String user) {  
    User user2 = new User();  
    user2.setId(2);  
    user2.setName("anand");  
    user2.setEmail("anand@amazon.com");  
  
    List<User> userList = new ArrayList<User>();  
    userList.add(user1);  
    userList.add(user2);  
    for (User users : userList) {  
  
        if(users.getName().contains(user)) {  
  
            return "<h1>ID:" + users.getId() + "</h1>"  
                +  
            "<h1>NAME:" + users.getName() + "</h1>"  
                +  
            "<h1>Email:" + users.getEmail() + "</h1>";  
  
        }  
  
    }  
    return "<h1>USER NOT FOUND</h1>";  
}
```

@RequestParam v/s @PathVariable

RequestParam	PathVariable
Request param extract the request parameters from URL(query string)	PathVariable extract request param from URI
Request param accept the value in key=value pair	Path variable accept only value

@PathVariable(Spring) = @PathParam (jaxrs,jersey)

@RequestParam(Spring) = @QueryParam(Jaxrs,Jersey)

Exception Handling in Rest api:

1.Custome Exception with @ResponseStatus.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)  
public class UserNotFoundException extends  
RuntimeException{  
  
    public UserNotFoundException(String message) {  
  
        super(message);  
    }  
  
}
```

UserNotFoundException.java

```

@GetMapping(path = "/user/{id}")
    public User getUser(@PathVariable int id) {
        User user1 = new User();
        user1.setId(1);
        user1.setName("vivek");
        user1.setEmail("vivek@amazoncom");

        List<User> userList = new ArrayList<User>();
        userList.add(user1);

        User u = userList.stream().filter(x-
>x.getId()==id).findAny().orElse(null);

        if(u==null)
            throw new UserNotFoundException("UserNot
Found with "+id);

        return u;
    }

```

UserController.java

2. ResponseEntity with HttpStatus Code.

```

@GetMapping(path = "/user/{id}")
    public ResponseEntity getUser(@PathVariable int id) {

        //create the user and assign to list like above

        User u = userList.stream().filter(x-
>x.getId()==id).findAny().orElse(null);
        if(u==null)
            return new ResponseEntity("User Not Found
with id "+id,HttpStatus.NOT_FOUND);

        return new ResponseEntity(u,HttpStatus.OK);
    }

```

Produces is argument for **@GetMapping** to architect the data exposed by api.

Ex: `@GetMapping(path = "/user/{id}", produces = "application/json")`

Consumes is an argument for **@PostMapping** to architect data to accept for api.

@PostMapping:

```
@PostMapping(path =  
    "/create", consumes=MediaType.dataformat)  
    public ResponseEntity<User> createUser(@RequestBody  
    User usr) {  
        userList.add(usr);  
        return new  
    ResponseEntity<User>(usr, HttpStatus.CREATED);  
    }
```

Consumes: how architect the data accept from client.

@PutMapping:

```
@PutMapping("/update/{id}")
public ResponseEntity updateUser(@PathVariable int
id,@RequestBody User user) {
    User u = null;
    for (User existinguser : userList) {
        if(existinguser.getId() == id) {

            existinguser.setName(user.getName());

            existinguser.setEmail(user.getEmail());
            u = existinguser;
        }
    }
    if(u==null)
        return new ResponseEntity("UerNot
Found",HttpStatus.NOT_FOUND);

    return new
ResponseEntity<User>(u,HttpStatus.CREATED);
}
```

Get	Post
Get call will transfer the data through Http Headers/RequestHeaders.	Post call will transfer the data through Http Body or Request Body.

Post	Put
Post call always create a new Object and insert.	Put call will retrieve the existing object and update it.

@DeleteMapping

```
@DeleteMapping("/delete/{id}")
    public String deleteUser(@PathVariable int id){
        User u = userList.stream().filter(x->x.getId() ==
id).findAny().orElse(null);
        if(u==null)
            throw new UserNotFoundException("User NOT
Found "+id);
        userList.remove(u);
        return "User Removed "+ id;
    }
```

To improve api readability its better to use class level annotation

@RequestMapping which will act as prefix to every call.

```
@RestController
@RequestMapping("/user")
public class UserController {

}
```

```
@RequestMapping("/user")
```

```
@GetMapping("/getall") @PostMapping(path = "/create") @PutMapping("/update/{id}") @DeleteMapping("/delete/{id}")
```

now api can access like

http://localhost:8089/user/getall

http://localhost:8089/user/create

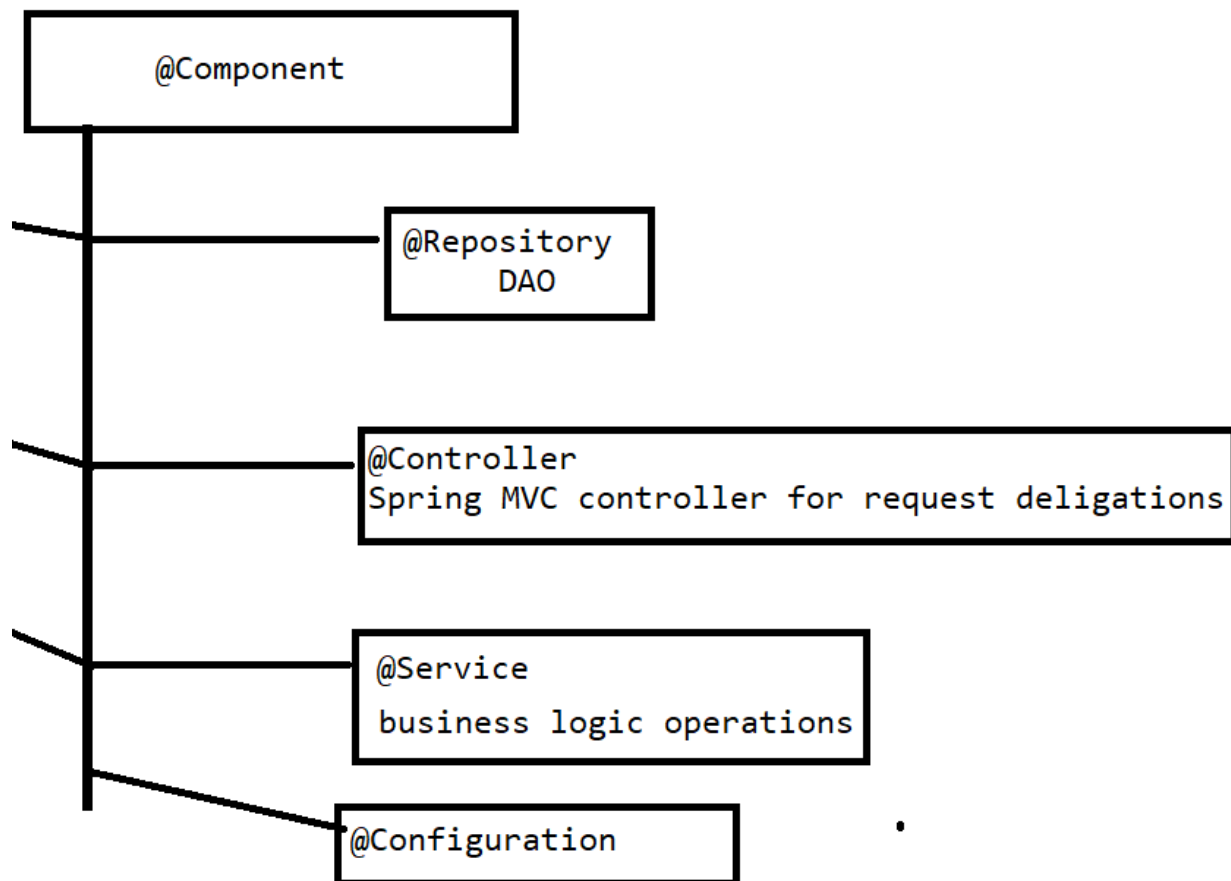
http://localhost:8089/user/update/1

http://localhost:8089/user/delete/2

Stereotype Annotations:

An annotation has set functionality which will drive an object according to the annotation specific.

EX:



@Component: Spring managed Bean

@Repository: persistence layer (DAO)

@Service: Business logic operations

@Controller: spring mvc configuration

@Configuration: to manage configurations for application/object.

SpringData:

Is a module spring boot which can utilize for database related operations.

Advantages:

- Less configuration.
- Auto detect driver class name based on JDBC url
- Supports hibernate dialect and remaining features like auto-create.

Spring Data offering 3 repositories which will reduce the burden of developer in db operations such as crud, paging and sorting.

JPA Repository:

Is the parent interface for all repositories which will give spring batch related operations.

CrudRepository:

Which is applicable to perform all the crud operations.

`saveAll()` to save the entire records at a time.

`findById(ID id)` to retrieve a record based on ID

`existsById(ID id)` to check wheter record available or not

`findAll()` find all related records

`findAllById(Iterable<ID> ids)` find all related id records

`count:` return no of records

`deleteById(ID id);` delete a record based on id

`delete()` to delete set of records.

PagingAndSortingRepository:

Applicable to add pagging and sorting features.

This repository take Pagable object to set records .

```
Pageable pageable =PageRequest.of(noOfPages,recordsPerPage);  
Page<User> userPage= userrepo.findAll(pageable);
```

```
@Repository  
public interface UserRepository extends  
PagingAndSortingRepository<User, Integer>{  
  
    public User getByName(String name);  
  
}
```

UserRepository.java

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
    private String email;
    private String adress;

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getAdress() {
        return adress;
    }
    public void setAdress(String adress) {
        this.adress = adress;
    }
}
```

User.java

```
@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired
    private UserRepository userrepo;

    @GetMapping("/getall")
    public ResponseEntity<?> getUser() {
        return new
ResponseEntity(userrepo.findAll(),HttpStatus.NOT_FOUND);
    }

    @GetMapping(path="/userpage/{pageNo}/{pageSize}")
    public List<User> getUserperPage(@PathVariable int
pageNo,@PathVariable int pageSize){
        Pageable pageable = PageRequest.of(pageNo,
pageSize);
        Page<User> userPage= userrepo.findAll(pageable);
        return userPage.toList();
    }

    @GetMapping(path = "/get/{id}")
    public ResponseEntity getUser(@PathVariable int id) {
        Optional<User> u = userrepo.findById(id);
        if(u.get()==null)
            throw new UserNotFoundException("User Not
found");
        return new ResponseEntity(u,HttpStatus.OK);
    }

    @GetMapping(path = "/getname/{name}")
    public ResponseEntity getUserByName(@PathVariable
String name) {
        User u = userrepo.getByName(name);
        if(u==null)
```

```

        throw new UserNotFoundException("User Not
found");
        return new ResponseEntity(u,HttpStatus.OK);
    }

    @PostMapping(path = "/create")
    public ResponseEntity<User> createUser(@RequestBody
User usr) {
        userrepo.save(usr);
        return new
ResponseEntity<User>(usr,HttpStatus.CREATED);
    }

    @PutMapping("/update/{id}")
    public ResponseEntity updateUser(@PathVariable int
id,@RequestBody User user) {
        Optional<User> u = userrepo.findById(id);
        User existinguser = u.get();
        if(user.getName()!=null)
            existinguser.setName(user.getName());
        if(user.getEmail()!=null)
            existinguser.setEmail(user.getEmail());
        if(user.getAdress()!=null)
            existinguser.setAdress(user.getAdress());
        userrepo.save(existinguser);
        return new
ResponseEntity(existinguser,HttpStatus.CREATED);
    }

    @DeleteMapping("/delete/{id}")
    public String deleteUser(@PathVariable int id){
        userrepo.deleteById(id);
        return "Deleted";
    }
}

```

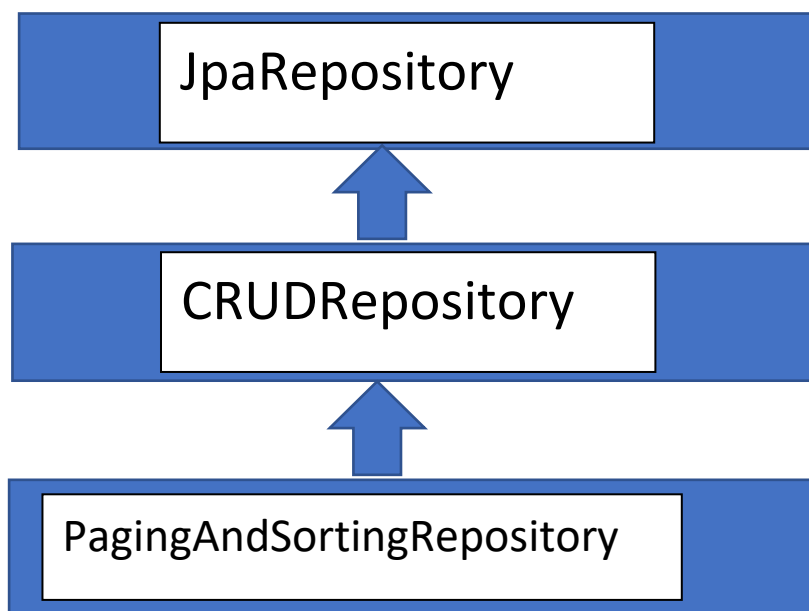
UserController.java

```
spring.h2.console.path=/h2-console
spring.datasource.url=jdbc:mysql://localhost:3306/training
#spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=admin
spring.jpa.database-
platform=org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.ddl-auto = update
server.port=8080
```

Application.properties

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

pom.xml



SpringCloud:

Service Registry and Discovery:

In the microservices world, Service Registry and Discovery plays an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers. In addition to that, in Cloud environments service instances may come up and go down anytime. So, we need some automatic service registration and discovery mechanism. We can use **Netflix Eureka** or **Consul** for Service Registry and Discovery. Here we are using SpringCloud Netflix Eureka for Service Registry and Discovery.

Add below dependency in pom.xml

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

We need to add **@EnableEurekaServer** annotation to make our SpringBoot application a Eureka Server based Service Registry.

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryserverApplication {

    public static void main(String[] args) {

        SpringApplication.run(DiscoveryserverApplication.class
, args);
    }

}
```

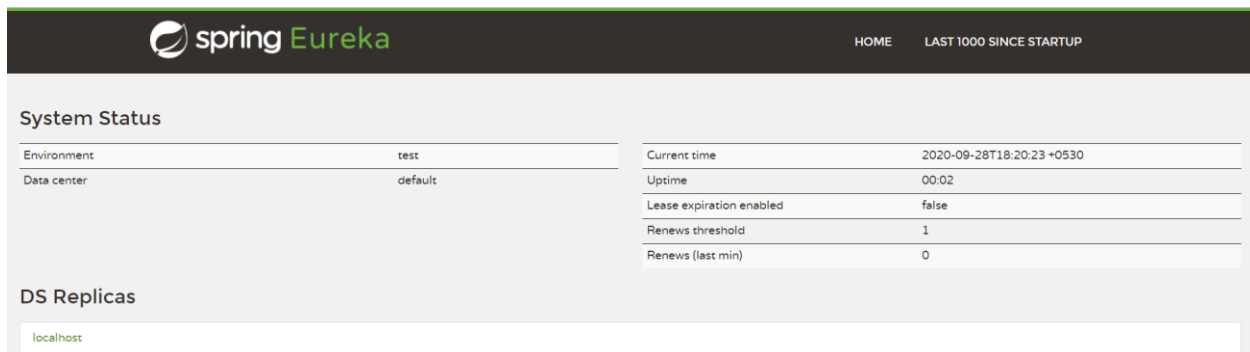
DiscoveryserverApplication.java

By default, each Eureka Server is also a Eureka client and needs at least one service URL to locate a peer. As we are going to have a single Eureka Server node (Standalone Mode), we are going to disable this client-side behavior by configuring the following properties in application.properties file.

```
spring.application.name=discoveryserver  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false  
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka
```

Netflix Eureka Service provides UI where we can see all the details about registered services.

Now run DiscoveryserverApplication and access [http://<host>:<port>/eureka](#) (Ex: [http://localhost:8080/eureka](#)) which will display the UI similar to below screenshot.



The screenshot shows the Spring Eureka web interface. The header includes the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into two sections: 'System Status' and 'DS Replicas'.

System Status

Environment	test	Current time	2020-09-28T18:20:23 +0530
Data center	default	Uptime	00:02
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

localhost

Now services can communicate without hard coding host and port.

Look at below code without Discovery server we are hard coding service host name and port.

We are taking two services

1.OrderService

2.UserService

Order service should send request to **User Service** to retrieve user details.

```
public OrderDetails getOrderById(int id) {  
  
    Orders order = orderRepo.findById(id).get();  
    orderDetails.setOrderid(order.getOrderid());  
  
    orderDetails.setOrderstatus(order.getOrderStatus());  
    User user =  
    restTemplate.getForObject("http://localhost:8081/user/"+ord  
er.getUserid(), User.class);  
  
    if(user!=null)  
        orderDetails.setUserdetails(user);  
    return orderDetails;  
  
}
```

Now we can convert User Service, Order Service as eureka clients then both services can access other service with service name without hard coding either host or port.

Add dependencies in client apps (Order, User services).

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

@EnableEurekaClient will convert service as Eureka client.

```
@SpringBootApplication
@EnableEurekaClient
public class UserappserviceApplication {

    public static void main(String[] args) {

        SpringApplication.run(UserappserviceApplication.class,
args);
    }

}
```

we just need to configure eureka.client.service-url.defaultZone property in application.properties to automatically register with the Eureka Server.

Add application name also for your service.

```
spring.application.name=OrderService
```

```
eureka.client.serviceUrl.defaultZone=http://localhost:8080/
eureka
```

Now check in Eureka dashboard you can see both services registered.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
ORDERSERVICE	n/a (1)	(1)	UP (1) - 192.168.2.8:OrderService:8081
USERSERVICE	n/a (1)	(1)	UP (1) - 192.168.2.8:UserService:8082

Now we can access with appname.

```
public OrderDetails getOrderById(int id) {  
  
    Orders order = orderRepo.findById(id).get();  
    orderDetails.setOrderid(order.getOrderid());  
  
    orderDetails.setOrderstatus(order.getOrderStatus());  
    User user =  
    restTemplate.getForObject(http://userserviceapi/+order.getUserId(), User.class);  
  
    if(user!=null)  
        orderDetails.setUserdetails(user);  
    return orderDetails;  
  
}
```

Zuul Proxy as API Gateway:

In microservices architecture, there could be several API services and few UI components that are talking to APIs. Instead of letting UI know about all our microservices details we can provide a unified proxy interface that will delegate the calls to various microservices based on URL pattern.

Spring Cloud provides Zuul proxy, similar to Nginx, that can be used to create API Gateway.

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
</dependency>
```

```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
public class ZuuldemoApplication {

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

}
```

Add below properties in application.properties.

```
spring.application.name=ZuulProxy
server.port=8087
zuul.prefix=/api

zuul.routes.userservice.path=/userservice/**
zuul.routes.userservice.serviceId=USERESERVICE

zuul.routes.orderservice.path=/orderservice/**
zuul.routes.orderservice.serviceId=ORDERSERVICE

eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.serviceUrl.defaultZone=http://localhost:8080/
eureka
```

Now from UI we can make a request to fetch products at <http://localhost:8080/api/orderservice/getorders>. By default, Zuul will strip the prefix and forward the request.

Circuit Breaker using Netflix Hystrix:

Netflix created Hystrix library implementing Circuit Breaker pattern. We can use Spring Cloud Netflix Hystrix Circuit Breaker to protect microservices from cascading failures.

Add below dependency.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

We can add another dependency to see visually circuit breakers.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-
dashboard
  </artifactId>
</dependency>
```

To enable Circuit Breaker add **@EnableCircuitBreaker** annotation on catalog-service entry-point class.

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@EnableHystrixDashboard
public class OrderserviceapiApplication {

    public static void main(String[] args) {

        SpringApplication.run(OrderserviceapiApplication.class
, args);
    }

}
```

Now we can use `@HystrixCommand` annotation on any method we want to apply timeout and fallback method.

```
@GetMapping("/greet")
    @HystrixCommand(fallbackMethod
    ="fallbackdemo",commandKey = "greet"
    )
    public String greet() {
        return orderService.greet();
    }

    public String fallbackdemo(){
        return "orderService fallback happend";
    }
```

You can see the HystrixDashBoard at <http://localhost:8081/hystrix>

To see Dashboard add a property in application.properties.

`hystrix.dashboard.proxyStreamAllowList=*`



Hystrix Dashboard

Cluster via Turbine (default cluster): <https://turbine-hostname:port/turbine.stream>

Cluster via Turbine (custom cluster): [https://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](https://turbine-hostname:port/turbine.stream?cluster=[clusterName])

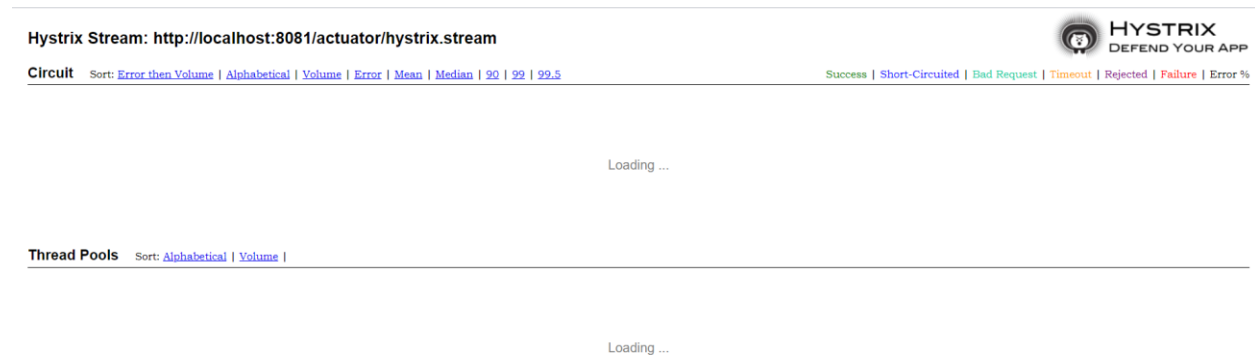
Single Hystrix App: <https://hystrix-app:port/actuator/hystrix.stream>

Delay: ms

Title:

To see the stream, enter url in first input field and click monitor.

<http://localhost:8081/actuator/hystrix.stream>.



Try to access <http://localhost:8081/greet> then you can see Hystrix spike as below.



Swagger:

we expose APIs as a back-end component for the front-end component or third-party app integrations.

In such a scenario, it is essential to have proper specifications for the back-end APIs. At the same time, the API documentation should be informative, readable, and easy to follow.

Moreover, reference documentation should simultaneously describe every change in the API. Accomplishing this manually is a tedious exercise, so automation of the process was inevitable.

Add dependency to get swagger libraries.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

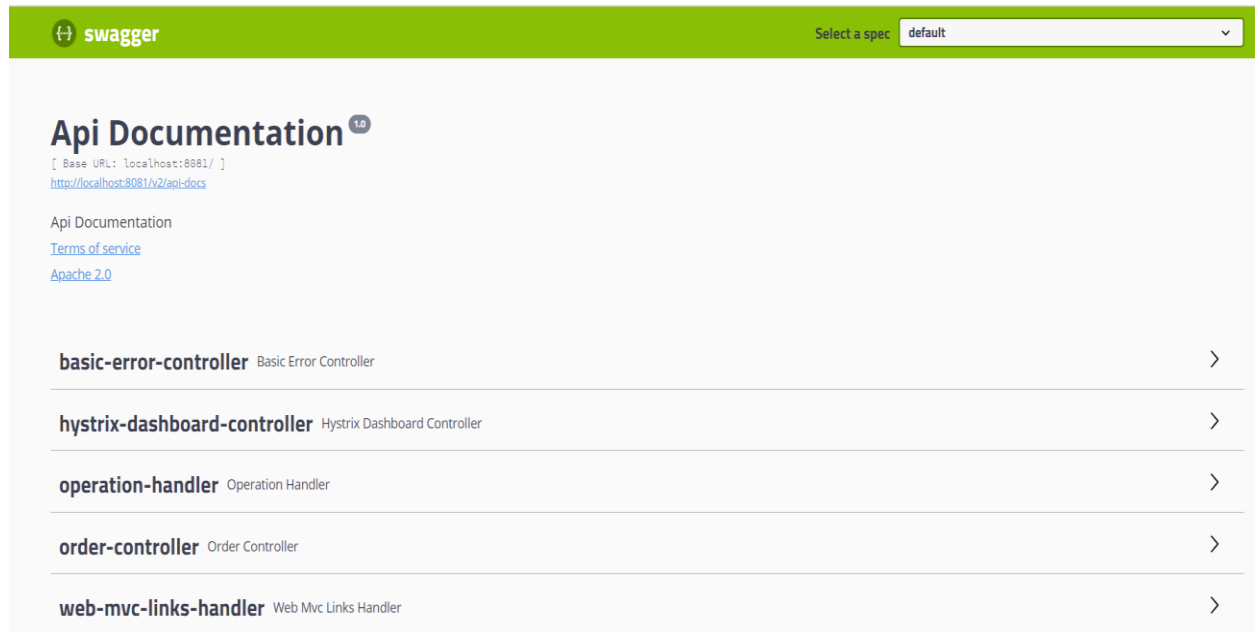
Add configuration.

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}
```


Then you can access service documentation from browser with below url:

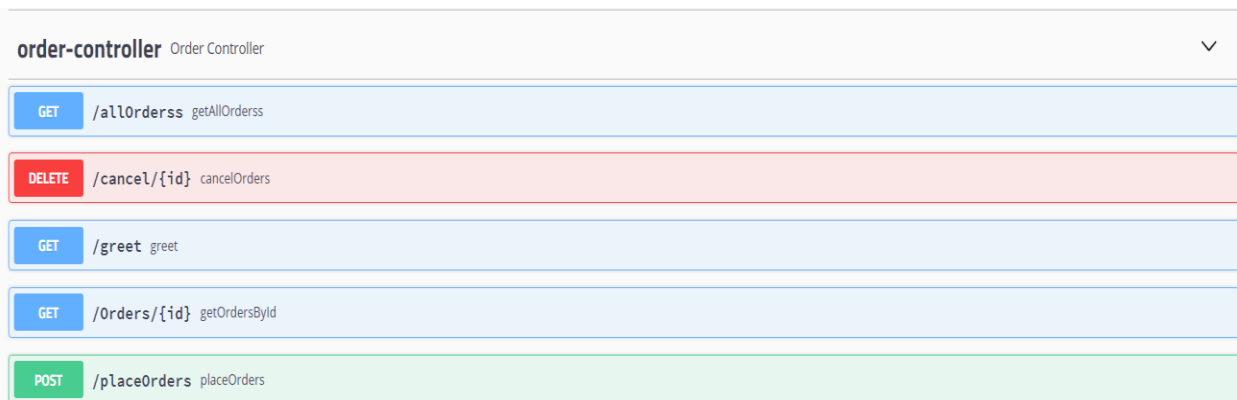
<http://localhost:8081/swagger-ui.html>



The screenshot shows the Swagger UI interface. At the top, there is a green header bar with the Swagger logo on the left and a dropdown menu labeled "Select a spec" with "default" selected on the right. Below the header, the main content area has a title "Api Documentation" with a version badge "1.0". Underneath the title, there is a base URL "[Base URL: 'localhost:8081/']" and a link "http://localhost:8081/v2/api-docs". Below this, there are links for "Api Documentation", "Terms of service", and "Apache 2.0". A list of controllers is displayed, each with a name, a description, and a right-pointing chevron icon:

- basic-error-controller** Basic Error Controller
- hystrix-dashboard-controller** Hystrix Dashboard Controller
- operation-handler** Operation Handler
- order-controller** Order Controller
- web-mvc-links-handler** Web Mvc Links Handler

We can expand any controller to check api documentation.



The screenshot shows the expanded "order-controller" section. The controller name "order-controller" and description "Order Controller" are at the top with a downward arrow icon. Below this, five API endpoints are listed, each with a colored header bar indicating the HTTP method:

- GET** `/allOrders` `getAllOrders`
- DELETE** `/cancel/{id}` `cancelOrders`
- GET** `/greet` `greet`
- GET** `/Orders/{id}` `getOrdersById`
- POST** `/placeOrders` `placeOrders`





























SpringSecurity:

Spring Security is a powerful and highly customizable authentication and access-control framework. Here we are building Spring MVC application that secures the page with a login form that is backed by a fixed list of users.

Add following dependencies from springboot initializer.

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-
jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
```

Project structure looks like below.

- ▼  securitydemo [boot]
 - >  Spring Elements
 - ▼  src/main/java
 - ▼  com.training.securitydemo
 - >  CreateUserDetails.java
 - >  SecuritydemoApplication.java
 - >  UserController.java
 - >  UserDetails.java
 - >  UserRepository.java
 - >  WebMvcConfig.java
 - >  WebSecurityConfig.java
 - ▼  src/main/resources
 -  static
 - ▼  templates
 -  details.html
 -  hello.html
 -  index.html
 -  login.html
 -  application.properties
 - >  src/test/java
 - >  JRE System Library [JavaSE-1.8]
 - >  Maven Dependencies
 - >  src
 - >  target
 -  HELP.md
 -  mvnw
 -  mvnw.cmd
 -  pom.xml

The web application is based on Spring MVC. As a result, you need to configure Spring MVC and set up view controllers to expose these templates. The following listing shows a class that configures Spring

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer{
    @Override
    public void addViewControllers(ViewControllerRegistry
registry) {
registry.addViewController("/home").setViewName("index");
registry.addViewController("/hello").setViewName("hello");
registry.addViewController("/info").setViewName("details");
registry.addViewController("/login").setViewName("login");

    }
}
```

WebMvcConfig.java

Suppose that you want to prevent unauthorized users from viewing the greeting page at /hello. As it is now, if visitors click the link on the home page, they see the greeting with no barriers to stop them. You need to add a barrier that forces the visitor to sign in before they can see that page.

You do that by configuring Spring Security in the application. If Spring Security is on the classpath, Spring Boot automatically secures all HTTP endpoints with “basic” authentication. However, you can further customize the security settings. The first thing you need to do is add Spring Security to the classpath.

The following security configuration ensures that only authenticated users can see the secret greeting

```

@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(HttpSecurity http) throws Exception
    {
        http
            .authorizeRequests()
                .antMatchers("/", "/home").permitAll()
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
                .and()
            .logout()
                .permitAll();
    }

    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        UserDetails user =
            User.withDefaultPasswordEncoder()
                .username("user")
                .password("password")
                .roles("USER")
                .build();

        return new InMemoryUserDetailsManager(user);
    }
}

```

WebSecurityConfig.java

The WebSecurityConfig class is annotated with @EnableWebSecurity to enable Spring Security's web security support and provide the Spring MVC integration. It also extends WebSecurityConfigurerAdapter and overrides a couple of its methods to set some specifics of the web security configuration.

The configure (HttpSecurity) method defines which URL paths should be secured and which should not. Specifically, the / and /home paths

are configured to not require any authentication. All other paths must be authenticated.

When a user successfully logs in, they are redirected to the previously requested page that required authentication. There is a custom /login page (which is specified by `loginPage()`), and everyone is allowed to view it.

The `userDetailsService()` method sets up an in-memory user store with a single user. That user is given a user name of `user`, a password of `password`, and a role of `USER`.

Spring security offers default login page if we want override then we can add our custom login page.

Now you need to create the login page. There is already a view controller for the login view, so you need only to create the login view itself, as the following listing shows:

Create `login.html`.

```
<html>
<title>Login</title>
</head>
<body>
<div th:if="${param.error}">
<h1 style="color: red">Invalid User/password</h1>
</div>
<div th:if="${param.logout}">
<h1 style="color: green">Logout Succes</h1>
</div>
<form th:action="@{/Login}" method="post">
UserName<input type="text" name="username">
Password<input type="password" name="password">
<input type="submit" value="LOGIN">
</form>
</body>
</html>
```

This Thymeleaf template presents a form that captures a username and password and posts them to /login. As configured, Spring Security provides a filter that intercepts that request and authenticates the user. If the user fails to authenticate, the page is redirected to /login?error, and your page displays the appropriate error message. Upon successfully signing out, your application is sent to /login?logout, and your page displays the appropriate success message.

Last, you need to provide the visitor a way to display the current user name and sign out. To do so, update the hello.html to say hello to the current user and contain a Sign Out form.

Create hello.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h1 th:inline="text">Hello
[[${#httpServletRequest.remoteUser}]]</h1>
<form th:action="@{/logout}" method="post">
<input type="submit" value="LogOut">
</form>
</body>
</html>
```

Hello.html

We display the username by using Spring Security's integration with `HttpServletRequest#getRemoteUser()`. The "Sign Out" form submits a

POST to /logout. Upon successfully logging out, it redirects the user to /login?logout.

Adding DB Authentication.

Instead of Inmemory we can authenticate external DB Adding following code in WebSecurityConfig class.

```
@Override
    protected void configure(AuthenticationManagerBuilder
authbuilder) throws Exception {

        authbuilder.jdbcAuthentication().passwordEncoder(new
BCryptPasswordEncoder())
            .dataSource(datasource)
            .usersByUsernameQuery("select
name,password,enabled from user_details where name = ?")
            .authoritiesByUsernameQuery("select name,role
from user_details where name = ?");
    }
```

Adding Security rest Controller:

Similarly, we can add security for rest controller also by using basic auth.

Develop a controller

```
@RestController
public class UserController {

    @GetMapping("/hello")
    public String greet() {
        return "Hello All from Controller";
    }

}
```


Add below configuration in Websecurityconfig.java

```
@Override
protected void configure(HttpSecurity http) throws
Exception{
    http.authorizeRequests()
        .antMatchers("/hello")
        .hasAnyAuthority("admin").
        anyRequest()
        .authenticated()
        .and()
        .httpBasic()
        .and()
        .csrf()
        .disable();
}
```

Test with postman.

Select Authorization tab and provide user name and password.

GET http://localhost:8080/hello Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

TYPE
Basic Auth

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Username: vivek

Password: *****

☐ Show Password

Response

Select Basic Auth from dropdown.

The screenshot shows a REST client interface with the following elements:

- Method:** GET
- URL:** http://localhost:8080/hello
- Buttons:** Send (blue), Sa (grey)
- Tabs:** Params, Authorization (selected), Headers (9), Body, Pre-request Script, Tests, Settings
- Authorization Tab:**
 - TYPE:** A dropdown menu is open, showing options: Basic Auth (selected), Inherit auth from parent, No Auth, API Key, Bearer Token, Digest Auth, OAuth 1.0, OAuth 2.0, Hawk Authentication, AWS Signature, NTLM Authentication [Beta], Akamai EdgeGrid.
 - Username:** vivek
 - Password:** *****
 - Show Password:** ☐
- Response Area:** Contains a rocket icon and the text "Hit Send to get a response".

If username and password valid then it will return data otherwise return 401-unauthorized if user don't have certain authority then 403 - forbidden.

Deploying Spring Boot Application in Docker:

Develop a basic application and build it.

Download and install docker.

Create docker file.

```
FROM openjdk:8
ADD target/dockerdemo.jar dockerdemo.jar
EXPOSE 7070
ENTRYPOINT ["java", "-jar", "dockerdemo.jar"]
```

Dockerfile

Dockerdemo.jar is application name.

Open command prompt and build image where Dockerfile exist.

```
docker build -f Dockerfile -t dockerdemo .  
docker run -p 8085:8085 dockerdemo
```

Deploying application in CloudFoundry.

Download and install Cloud foundry CLI.

Create an account in cloudfoundry.

Login with below command.

```
cf login -a api.run.pivotal.io -u demo@gmail.com -p 007
```

Create manifest.yml in application.

```
applications:  
- name: dockerdemoapp  
  path: target/dockerdemo.jar  
  domain: cfapps.io  
  instance: 1
```

Manifest.yml

Then cf push.

That's it you can see app in clod foundry.

Miscellaneous:

@Component

 @Controller

 @RestController(@Controller+@ResponseBody)

 @Configuration

 @Bean

@RequestMapping

 @GetMapping

 @PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

@PathVariable

@RequestParam

@RequestBody

@ResponseStatus

@Autowired

@SpringBootApplication

produces

consumes

ResponseEntity

HttpStatus

CustomException with ResponseEntity

ResponseEntity with statusCode