learning-zone / **spring-interview-questions** Public

500+ Spring-Boot Interview Questions

Unlicense license

238 stars 144 forks

Star Watch

Code Issues Pull requests Actions Projects Wiki Security Insig

spring

learning-zone Merge pull request #3 from Ruthwik/pr ... on Jul 23, 2021 562

View code

README.md

Spring-Boot & Microservices Interview Questions

Click ★ if you like the project. Pull Request are highly appreciated.

Table of Contents

Q. *Spring Boot RESTful Web Service example?*

Step 01: pom.xml Settings

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.springexample</groupId>
    <artifactId>SpringBootCrudRestful</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>SpringBootCrudRestful</name>
    <description>Spring Boot + Restful</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

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

        <dependency>
            <groupId>com.fasterxml.jackson.dataformat</groupId>
            <artifactId>jackson-dataformat-xml</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
```

```
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
            </plugins>
        </build>

    </project>
```

Step 02: SpringBootCrudRestfulApplication.java

```
    import org.springframework.boot.SpringApplication;
    import org.springframework.boot.autoconfigure.SpringBootApplication;

    @SpringBootApplication
    public class SpringBootCrudRestfulApplication {

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

Step 03: Employee.java

```
    public class Employee {

        private String empNo;
        private String empName;
        private String position;

        public Employee() { }

        public Employee(String empNo, String empName, String position) {
            this.empNo = empNo;
            this.empName = empName;
            this.position = position;
        }

        public String getEmpNo() {
            return empNo;
        }

        public void setEmpNo(String empNo) {
            this.empNo = empNo;
```

```
    }

    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }

}
```

Step 04: EmployeeDAO.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springexample.sbcrudrestful.model.Employee;
import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDAO {

    private static final Map<String, Employee> empMap = new HashMap<String, Employee

    static {
        initEmps();
    }

    private static void initEmps() {
        Employee emp1 = new Employee("E01", "Smith", "Clerk");
        Employee emp2 = new Employee("E02", "Allen", "Salesman");
        Employee emp3 = new Employee("E03", "Jones", "Manager");

        empMap.put(emp1.getEmpNo(), emp1);
        empMap.put(emp2.getEmpNo(), emp2);
        empMap.put(emp3.getEmpNo(), emp3);
```

```
    }

    public Employee getEmployee(String empNo) {
        return empMap.get(empNo);
    }

    public Employee addEmployee(Employee emp) {
        empMap.put(emp.getEmpNo(), emp);
        return emp;
    }

    public Employee updateEmployee(Employee emp) {
        empMap.put(emp.getEmpNo(), emp);
        return emp;
    }

    public void deleteEmployee(String empNo) {
        empMap.remove(empNo);
    }

    public List<Employee> getAllEmployees() {
        Collection<Employee> c = empMap.values();
        List<Employee> list = new ArrayList<Employee>();
        list.addAll(c);
        return list;
    }

}
```

Step 05: MainRESTController.java

```
import java.util.List;

import org.springexample.sbcrudrestful.dao.EmployeeDAO;
import org.springexample.sbcrudrestful.model.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MainRESTController {
```

```
@Autowired
private EmployeeDAO employeeDAO;

@RequestMapping("/")
@ResponseBody
public String welcome() {
    return "Welcome to RestTemplate Example.";
}

@RequestMapping(value = "/employees", //
        method = RequestMethod.GET, //
        produces = { MediaType.APPLICATION_JSON_VALUE, //
                MediaType.APPLICATION_XML_VALUE })
@ResponseBody
public List<Employee> getEmployees() {
    List<Employee> list = employeeDAO.getAllEmployees();
    return list;
}

@RequestMapping(value = "/employee/{empNo}", //
        method = RequestMethod.GET, //
        produces = { MediaType.APPLICATION_JSON_VALUE, //
                MediaType.APPLICATION_XML_VALUE })
@ResponseBody
public Employee getEmployee(@PathVariable("empNo") String empNo) {
    return employeeDAO.getEmployee(empNo);
}

@RequestMapping(value = "/employee", //
        method = RequestMethod.POST, //
        produces = { MediaType.APPLICATION_JSON_VALUE, //
                MediaType.APPLICATION_XML_VALUE })
@ResponseBody
public Employee addEmployee(@RequestBody Employee emp) {

    System.out.println("(Service Side) Creating employee: " + emp.getEmpNo());

    return employeeDAO.addEmployee(emp);
}

@RequestMapping(value = "/employee", //
        method = RequestMethod.PUT, //
        produces = { MediaType.APPLICATION_JSON_VALUE, //
                MediaType.APPLICATION_XML_VALUE })
@ResponseBody
public Employee updateEmployee(@RequestBody Employee emp) {

    System.out.println("(Service Side) Editing employee: " + emp.getEmpNo());
```

```
        return employeeDAO.updateEmployee(emp);
    }

    @RequestMapping(value = "/employee/{empNo}", //
            method = RequestMethod.DELETE, //
            produces = { MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML
    @ResponseBody
    public void deleteEmployee(@PathVariable("empNo") String empNo) {

        System.out.println("(Service Side) Deleting employee: " + empNo);

        employeeDAO.deleteEmployee(empNo);
    }
}
```

Step 06: Run and Test the application

```
// Get all the employees details
http://localhost:8080/employees
http://localhost:8080/employees.json
http://localhost:8080/employees.xml


// Get the employee based in employee-id
http://localhost:8080/employee/E01
http://localhost:8080/employee/E01.xml
http://localhost:8080/employee/E01.json
```

↥ back to top

Q. *Spring Boot Program to Connect with databases?*

Step 01: application.properties Settings

```
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb
spring.datasource.username=root
spring.datasource.password=mysql
spring.jpa.hibernate.ddl-auto=create-drop
```

Step 02: SpringBootJdbcApplication.java

```
package com.learningzone;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringBootJdbcApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootJdbcApplication.class, args);
    }
}
```

Step 03: SpringBootJdbcController.java

```
package com.learningzone;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class SpringBootJdbcController {
    @Autowired
    JdbcTemplate jdbc;
    @RequestMapping("/insert")
    public String index(){
        jdbc.execute("insert into user(name, email) values('Pradeep Kumar','pradeep.
        return "Record inserted Successfully";
    }
}
```

↥ back to top

Q. *Spring Boot program for file upload and download?*

Step 01: Configuring Server and File Storage Properties

```
#src/main/resources/application.properties

## MULTIPART (MultipartProperties)
# Enable multipart uploads
spring.servlet.multipart.enabled=true
# Threshold after which files are written to disk.
spring.servlet.multipart.file-size-threshold=2KB
# Max file size.
spring.servlet.multipart.max-file-size=200MB
# Max Request Size
```

```
spring.servlet.multipart.max-request-size=215MB

## File Storage Properties
# All files uploaded through the REST API will be stored in this directory
file.upload-dir=/Users/files/uploads
```

Step 02: Automatically binding properties to a POJO class

```
package com.example.filedemo.property;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "file")
public class FileStorageProperties {
    private String uploadDir;

    public String getUploadDir() {
        return uploadDir;
    }

    public void setUploadDir(String uploadDir) {
        this.uploadDir = uploadDir;
    }
}
```

Step 03: Enable Configuration Properties

```
/* src/main/java/com/example/filedemo/FileDemoApplication.java */
package com.example.filedemo;

import com.example.filedemo.property.FileStorageProperties;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.properties.EnableConfigurationProperties;

@SpringBootApplication
@EnableConfigurationProperties({
        FileStorageProperties.class
})
public class FileDemoApplication {

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

Step 04: Writing APIs for File Upload and Download

```
package com.example.filedemo.controller;

import com.example.filedemo.payload.UploadFileResponse;
import com.example.filedemo.service.FileStorageService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

@RestController
public class FileController {

    private static final Logger logger = LoggerFactory.getLogger(FileController.clas

    @Autowired
    private FileStorageService fileStorageService;

    @PostMapping("/uploadFile")
    public UploadFileResponse uploadFile(@RequestParam("file") MultipartFile file) {
        String fileName = fileStorageService.storeFile(file);

        String fileDownloadUri = ServletUriComponentsBuilder.fromCurrentContextPath(
                .path("/downloadFile/")
                .path(fileName)
                .toUriString();

        return new UploadFileResponse(fileName, fileDownloadUri,
                file.getContentType(), file.getSize());
    }

    @PostMapping("/uploadMultipleFiles")
    public List<UploadFileResponse> uploadMultipleFiles(@RequestParam("files") Multi
        return Arrays.asList(files)
                .stream()
                .map(file -> uploadFile(file))
```

```
                .collect(Collectors.toList());
    }

    @GetMapping("/downloadFile/{fileName:.+}")
    public ResponseEntity<Resource> downloadFile(@PathVariable String fileName, Http
        // Load file as Resource
        Resource resource = fileStorageService.loadFileAsResource(fileName);

        // Try to determine file's content type
        String contentType = null;
        try {
            contentType = request.getServletContext().getMimeType(resource.getFile()
        } catch (IOException ex) {
            logger.info("Could not determine file type.");
        }

        // Fallback to the default content type if type could not be determined
        if(contentType == null) {
            contentType = "application/octet-stream";
        }

        return ResponseEntity.ok()
                .contentType(MediaType.parseMediaType(contentType))
                .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
                .body(resource);
    }
}
```

Step 05: UploadFileResponse

```
package com.example.filedemo.payload;

public class UploadFileResponse {
    private String fileName;
    private String fileDownloadUri;
    private String fileType;
    private long size;

    public UploadFileResponse(String fileName, String fileDownloadUri, String fileTy
        this.fileName = fileName;
        this.fileDownloadUri = fileDownloadUri;
        this.fileType = fileType;
        this.size = size;
    }

        // Getters and Setters (Omitted for brevity)
}
```

Step 06: Service for Storing Files in the FileSystem and retrieving them

```
package com.example.filedemo.service;

import com.example.filedemo.exception.FileStorageException;
import com.example.filedemo.exception.CustomFileNotFoundException;
import com.example.filedemo.property.FileStorageProperties;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.core.io.UrlResource;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
import org.springframework.web.multipart.MultipartFile;
import java.io.IOException;
import java.net.MalformedURLException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

@Service
public class FileStorageService {

    private final Path fileStorageLocation;

    @Autowired
    public FileStorageService(FileStorageProperties fileStorageProperties) {
        this.fileStorageLocation = Paths.get(fileStorageProperties.getUploadDir())
                .toAbsolutePath().normalize();

        try {
            Files.createDirectories(this.fileStorageLocation);
        } catch (Exception ex) {
            throw new FileStorageException("Could not create the directory where the
        }
    }

    public String storeFile(MultipartFile file) {
        // Normalize file name
        String fileName = StringUtils.cleanPath(file.getOriginalFilename());

        try {
            // Check if the file's name contains invalid characters
            if(fileName.contains("..")) {
                throw new FileStorageException("Sorry! Filename contains invalid pat
            }
```

```
            // Copy file to the target location (Replacing existing file with the sa
            Path targetLocation = this.fileStorageLocation.resolve(fileName);
            Files.copy(file.getInputStream(), targetLocation, StandardCopyOption.REP

            return fileName;
        } catch (IOException ex) {
            throw new FileStorageException("Could not store file " + fileName + ". P
        }
    }

    public Resource loadFileAsResource(String fileName) {
        try {
            Path filePath = this.fileStorageLocation.resolve(fileName).normalize();
            Resource resource = new UrlResource(filePath.toUri());
            if(resource.exists()) {
                return resource;
            } else {
                throw new CustomFileNotFoundException("File not found " + fileName);
            }
        } catch (MalformedURLException ex) {
            throw new CustomFileNotFoundException("File not found " + fileName, ex);
        }
    }
}
```

Step 07: FileStorageException

```
package com.example.filedemo.exception;

public class FileStorageException extends RuntimeException {
    public FileStorageException(String message) {
        super(message);
    }

    public FileStorageException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Step 08: CustomFileNotFoundException

```
package com.example.filedemo.exception;

import org.springframework.http.HttpStatus;
```

```
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class CustomFileNotFoundException extends RuntimeException {
    public CustomFileNotFoundException(String message) {
        super(message);
    }

    public CustomFileNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Step 09: Running the Application and Testing the APIs via Postman

```
mvn spring-boot:run
```

↥ back to top

Q. *Spring Boot program for Sending Email?*

Step 01: pom.xml Settings

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-boot-send-email</artifactId>
    <packaging>jar</packaging>
    <name>Spring Boot Send Email</name>
    <url>https://www.springboot.com</url>
    <version>1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.2.RELEASE</version>
    </parent>

    <properties>
        <java.version>1.8</java.version>
    </properties>
```

```
    <dependencies>

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

        <!-- send email -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-mail</artifactId>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <!-- Package as an executable jar/war -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
            </plugin>

        </plugins>
    </build>
</project>
```

Step 02: application.properties Settings

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=pradeep.vwa@gmail.com
spring.mail.password=*****

# Other properties
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=5000
spring.mail.properties.mail.smtp.writetimeout=5000

# TLS , port 587
```

```
spring.mail.properties.mail.smtp.starttls.enable=true

# SSL, post 465
#spring.mail.properties.mail.smtp.socketFactory.port = 465
#spring.mail.properties.mail.smtp.socketFactory.class =
javax.net.ssl.SSLSocketFactory
```

Step 03: Application.java

```
package com.springtutorial;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.core.io.ClassPathResource;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;
import java.io.IOException;

@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    private JavaMailSender javaMailSender;

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

    @Override
    public void run(String... args) {

        System.out.println("Sending Email...");
        try {
            sendEmail();
            //sendEmailWithAttachment();

        } catch (MessagingException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("Done");
```

```
    }

    void sendEmail() {

        SimpleMailMessage msg = new SimpleMailMessage();
        msg.setTo("pradeep.vwa@gmail.com", "pradeep.vwa@gmail.com");
        msg.setSubject("Testing from Spring Boot");
        msg.setText("Hello World \n Spring Boot Email");

        javaMailSender.send(msg);
    }

    void sendEmailWithAttachment() throws MessagingException, IOException {

        MimeMessage msg = javaMailSender.createMimeMessage();

        // true = multipart message
        MimeMessageHelper helper = new MimeMessageHelper(msg, true);
        helper.setTo("pradeep.vwa@gmail.com");
        helper.setSubject("Testing from Spring Boot");
        helper.setText("<h1>Check attachment for image!</h1>", true);
        helper.addAttachment("my_photo.png", new ClassPathResource("android.png"));

        javaMailSender.send(msg);
    }
}
```

↥ back to top

Q. *What is difference between spring and spring boot?*

Basis of Differentiation	Spring	Spring Boot
Configuration	In order to design any Spring based application, the developer has to take recourse to the annual setup feature on the Hibernate data source. Session Factory, entity Manager, Transaction Management, etc. have to be configured as well.	The common set up and features of Spring Boot do not have to be designed by the developer individually. The Spring Boot Configuration annotation is well-equipped to handle everything at the time of deployment.

Basis of Differentiation	Spring	Spring Boot
XML	In Spring MVC applications, some XML definitions are to be managed mandatorily.	In the configuration of Spring Boot applications, nothing has to be managed manually. The annotations are capable of managing all that is needed.
Controlling	As the configuration can be easily handled manually, Spring or Spring Boot need not load some unwanted default features for specific applications.	In Spring Boot, the controls are automatically handled during the default loading part. As such, developers do not have the option of not loading unusable components belonging to the default Spring Boot features.
Use	Better to use if characteristics or application type are purely defined.	Better to use in cases where the application type of functionality of future use is not properly defined. As the task of integrating any Spring-specific feature is auto-configured in this case, there is no necessity of any additional configuration.

↥ back to top

Q. *Explain types of spring bean scopes?*

The core of spring framework is it's bean factory and mechanisms to create and manage such beans inside Spring container. The beans in spring container can be created in six scopes i.e. singleton, prototype, request, session, application and websocket. They are called spring bean scopes.

SCOPE	DESCRIPTION
singleton (default)	Single bean object instance per spring IoC container

SCOPE	DESCRIPTION
prototype	Opposite to singleton, it produces a new instance each and every time a bean is requested.
request	A single instance will be created and available during complete lifecycle of an HTTP request. Only valid in web-aware Spring ApplicationContext.
session	A single instance will be created and available during complete lifecycle of an HTTP Session. Only valid in web-aware Spring ApplicationContext.
application	A single instance will be created and available during complete lifecycle of ServletContext. Only valid in web-aware Spring ApplicationContext.
websocket	A single instance will be created and available during complete lifecycle of WebSocket. Only valid in web-aware Spring ApplicationContext.

1. singleton scope

singleton is default bean scope in spring container. It tells the container to create and manage only one instance of bean class, per container. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached instance.

Example of singleton scope bean using Java config –

```
@Component
// This statement is redundant - singleton is default scope
@Scope("singleton")  // This statement is redundant
public class BeanClass {

}
```

Example of singleton scope bean using XML config –

```
<!-- To specify singleton scope is redundant -->
<bean id="beanId" class="com.springexample.BeanClass" scope="singleton" />
// or
<bean id="beanId" class="com.springexample.BeanClass" />
```

2. prototype scope

prototype scope results in the creation of a new bean instance every time a request for the bean is made by application code.

Java config example of prototype bean scope –

```
@Component
@Scope("prototype")
public class BeanClass {
}
```

XML config example of prototype bean scope –

```
<bean id="beanId" class="com.springexample.BeanClass" scope="prototype" />
```

3. request scope

In request scope, container creates a new instance for each and every HTTP request. So, if server is currently handling 5 requests, then container can have at most 5 individual instances of bean class.

Java config example of request bean scope –

```
@Component
@Scope("request")
public class BeanClass {
}

// or

@Component
@RequestScope
public class BeanClass {
}
```

XML config example of request bean scope –

```
<bean id="beanId" class="com.springexample.BeanClass" scope="request" />
```

4. session scope

In session scope, container creates a new instance for each and every HTTP session. So, if server has 10 active sessions, then container can have at most 10 individual instances of bean class. All HTTP requests within single session lifetime will have access to same single bean instance in that session scope.

Java config example of session bean scope –

```
@Component
@Scope("session")
public class BeanClass {
}

// or

@Component
@SessionScope
public class BeanClass {
}
```

XML config example of session bean scope –

```
<bean id="beanId" class="com.springexample.BeanClass" scope="session" />
```

5. application scope

In application scope, container creates one instance per web application runtime. It is almost similar to singleton scope, with only two differences i.e.

- application scoped bean is singleton per ServletContext, whereas singleton scoped bean is singleton per ApplicationContext. Please note that there can be multiple application contexts for single application.
- application scoped bean is visible as a ServletContext attribute.

Java config example of application bean scope –

```
@Component
@Scope("application")
public class BeanClass {
}

// or

@Component
@ApplicationScope
public class BeanClass {
}
```

XML config example of application bean scope –

```
<bean id="beanId" class="com.springexample.BeanClass" scope="application" />
```

6. websocket scope

The WebSocket Protocol enables two-way communication between a client and a remote host that has opted-in to communication with client. WebSocket Protocol provides a single TCP connection for traffic in both directions.

Java config example of websocket bean scope –

```
@Component
@Scope("websocket")
public class BeanClass {
}
```

XML config example of websocket bean scope –

```
<bean id="beanId" class="com.springexample.BeanClass" scope="websocket" />
```

↥ back to top

Q. *What is AOP? what does spring AOP provide?*

Spring AOP enables Aspect-Oriented Programming in spring applications. In AOP, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (often termed crosscutting concerns).

AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations. It makes easy to maintain code in the present and future as well.

- **Aspect Oriented Programming Core Concepts**

1. **Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management.
2. **Join Point**: A join point is the specific point in the application such as method execution, exception handling, changing object variable values etc. In Spring AOP a join points is always the execution of a method.

3. **Advice**: Advices are actions taken for a particular join point. In terms of programming, they are methods that gets executed when a certain join point with matching pointcut is reached in the application.

4. **Pointcut**: Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.

5. **Weaving**: It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

- **Types of Advices**

1. **Before Advice**: These advices runs before the execution of join point methods. We can use @Before annotation to mark an advice type as Before advice.

2. **After returning advice**: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

3. **After throwing advice**: Advice to be executed if a method exits by throwing an exception.

4. **After advice**: Advice to be executed regardless of the means by which a join point exits.

5. **Around advice**: Around advice can perform custom behavior before and after the method invocation. This type of advice is used where we need frequent access to a method or database like- caching.

Example: Types of Advices

```
/**
* AOP program to illustrate types of Advices
*
*//
@Aspect
class Logging {

    // **Before**
    @Before("execution(public void com.aspect.ImplementAspect.aspectCall())")
    public void loggingAdvice1() {
        System.out.println("Before advice is executed");
    }
```

```
    // **After**
    @After("execution(public void com.aspect.ImplementAspect.aspectCall())")
    public void loggingAdvice2() {
        System.out.println("Running After Advice.");
    }

    // **Around**
    @Around("execution(public void com.aspect.ImplementAspect.myMethod())")
    public void loggingAdvice3() {
        System.out.println("Before and After invoking method myMethod");
    }

    // **AfterThrowing**
    @AfterThrowing("execution(" public void com.aspect.ImplementAspect.aspectCall())
    public void loggingAdvice4() {
        System.out.println("Exception thrown in method");
    }

    // **AfterRunning**
    @AfterReturning("execution(public void com.aspect.ImplementAspect.myMethod())")
    public void loggingAdvice5() {
        System.out.println("AfterReturning advice is run");
    }
}
```

Example: JoinPoints

```
/**
* AOP program to illustrate JoinPoints
*
**/

@Aspect
class Logging {

    // Passing a JoinPoint Object into parameters of the method
    // with the annotated advice enables to print the information

    @Before("execution(public void com.aspect.ImplementAspect.aspectCall())")
    public void loggingAdvice1(JoinPoint joinpoint) {
        System.out.println("Before advice is executed");
        System.out.println(joinpoint.toString());
    }
}
```

Example: PointCuts

```
/**
* AOP program to illustrate PointCuts
*
**/
@Aspect
class Logging {

    @Pointcut("execution(public void com.aspect.ImplementAspect.aspectCall())")
    public void pointCut() {
    }

    // pointcut() is used to avoid repeatition of code
    @Before("pointcut()")
    public void loggingAdvice1() {
        System.out.println("Before advice is executed");
    }
}
```

↥ back to top

Q. *What is the difference between @Component, @Repository & @Service annotations in Spring?*

- **@Component** This is a general-purpose stereotype annotation indicating that the class is a spring component.

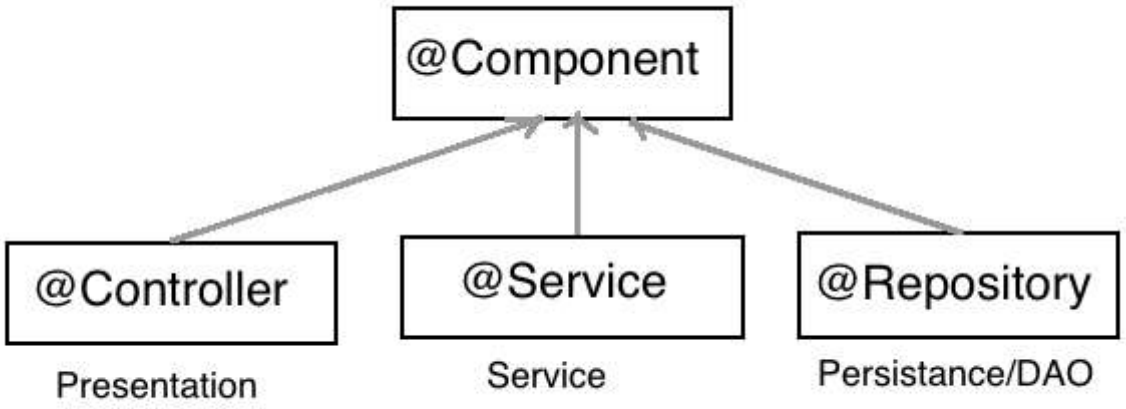
```
@Component
public @interface Service {
    ….
}
```

- **@Repository** This is to indicate that the class defines a database repository.

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostP
```

- **@Controller** This indicate that the annotate classes at presentation layers level, mainly used in Spring MVC.

- **@Service** `@Service` beans hold the business logic and call methods in the repository layer.



↥ back to top

Q. *How to do SSO implementation using Spring Boot?*

Single sign-on (or SSO) allow users to use a single set of credentials to login into multiple related yet independent web applications. SSO is achieved by implementing a centralised login system that handles authentication of users and share that information with applications that need that data.

Example: **Simple Single Sign-On with Spring Security OAuth2**

Step 01: Maven Dependencies (pom.xml)

```
<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.security.oauth.boot</groupId>
    <artifactId>spring-security-oauth2-autoconfigure</artifactId>
    <version>2.0.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.thymeleaf.extras</groupId>
        <artifactId>thymeleaf-extras-springsecurity4</artifactId>
    </dependency>
```

Step 02: Security Configuration

```
@Configuration
@EnableOAuth2Sso
public class UiSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/**")
          .authorizeRequests()
          .antMatchers("/", "/login**")
          .permitAll()
          .anyRequest()
          .authenticated();
    }
}
```

Step 03: OAuth Configuration

```
@SpringBootApplication
@EnableResourceServer
public class AuthorizationServerApplication extends SpringBootServletInitializer {
    public static void main(String[] args) {
        SpringApplication.run(AuthorizationServerApplication.class, args);
    }
}
```

Step 04: Security Configuration

```
@Configuration
@Order(1)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.requestMatchers()
          .antMatchers("/login", "/oauth/authorize")
```

```
            .and()
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin().permitAll();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("john")
            .password(passwordEncoder().encode("123"))
            .roles("USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}
```

Step 05: User Endpoint

```
@RestController
public class UserController {
    @GetMapping("/user/me")
    public Principal user(Principal principal) {
        return principal;
    }
}
```

↥ back to top

Q. *What is difference between DI and IOC in spring?*

- **DI(Dependency Injection)**

Dependency injection is a pattern used to create instances of objects that other objects rely upon without knowing at compile time which class will be used to provide that functionality or simply the way of injecting properties to an object is called dependency injection.

There are 3 types of Dependency injection

1. Constructor Injection
2. Setter/Getter Injection
3. Interface Injection
   Spring support only Constructor Injection and Setter/Getter Injection.

- **IOC(Inversion Of Control)**

Giving control to the container to create and inject instances of objects that your application depend upon, means instead of you are creating an object using the new operator, let the container do that for you. Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality

The two concepts work together in this way to allow for much more flexible, reusable, and encapsulated code to be written. As such, they are important concepts in designing object-oriented solutions.

↥ back to top

Q. *What is main advantage of RESTful implementation over SOAP?*

SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are both web service communication protocols. In addition to using HTTP for simplicity, REST offers a number of other benefits over SOAP:

SOAP

- SOAP is a protocol.
- SOAP stands for Simple Object Access Protocol.
- SOAP can't use REST because it is a protocol.
- SOAP uses services interfaces to expose the business logic.
- SOAP defines standards to be strictly followed.
- SOAP requires more bandwidth and resource than REST.
- SOAP defines its own security.
- SOAP permits XML data format only.
- SOAP is less preferred than REST.

REST

- REST is an architectural style.

- REST stands for Representational State Transfer.
- REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
- REST uses URI to expose business logic.
- REST does not define too much standards like SOAP.
- REST requires less bandwidth and resource than SOAP.
- RESTful web services inherits security measures from the underlying transport.
- REST permits different data format such as Plain text, HTML, XML, JSON etc.
- REST more preferred than SOAP.

↥ back to top

Q. *What is Spring Cloud?*

Spring Cloud, in microservices, is a system that provides integration with external systems. It is a short-lived framework that builds an application, fast. Being associated with the finite amount of data processing, it plays a very important role in microservice architectures.

For typical use cases, Spring Cloud provides the out of the box experiences and a sets of extensive features mentioned below:

- Versioned and distributed configuration.
- Discovery of service registration.
- Service to service calls.
- Routing.
- Circuit breakers and load balancing.
- Cluster state and leadership election.
- Global locks and distributed messaging.

↥ back to top

Q. *What is Role of Actuator in Spring Boot?*

It helps to access the current state of an application that is running in a production environment. There are multiple metrics which can be used to check the current state. They also provide endpoints for RESTful web services which can be simply used to check the different metrics.

↥ back to top

Q. *Which Embedded Containers are Supported by Spring Boot?*

Spring Boot contains Jetty, Tomcat, and Undertow servers, all of which are embedded.

- **Jetty –** Used in a wide number of projects, Eclipse Jetty can be embedded in framework, application servers, tools, and * clusters.
- **Tomcat –** Apache Tomcat is an open source JavaServer Pages implementation which works well with embedded systems.
- **Undertow –** A flexible and prominent web server that uses small single handlers to develop a web server.

↥ back to top

Q. *What are the advantages of using Spring Cloud?*

When developing distributed microservices with Spring Boot we face the following issues-

- **Complexity associated with distributed systems-**
  This overhead includes network issues, Latency overhead, Bandwidth issues, security issues.
- **Service Discovery-**
  Service discovery tools manage how processes and services in a cluster can find and talk to one another. It involves a directory of services, registering services in that directory, and then being able to lookup and connect to services in that directory.
- **Redundancy-**
  Redundancy issues in distributed systems.
- **Loadbalancing-**
  Load balancing improves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives.
- **Performance issues-**
  Performance issues due to various operational overheads.
- **Deployment complexities-**
  Requirement of Devops skills.

↥ back to top

Q. *How to achieve server side load balancing using Spring Cloud?*

Server side load balancing can be achieved using `Netflix Zuul` . Zuul is a JVM based router and server side load balancer by Netflix. It provides a single entry to our system, which allows a browser, mobile app, or other user interface to consume services from multiple hosts without managing cross-origin resource sharing (CORS) and authentication for each one. We can integrate Zuul with other Netflix projects like Hystrix for fault tolerance and Eureka for service discovery, or use it to manage routing rules, filters, and load balancing across your system.

↥ back to top

Q. *What are the advantages of using Spring Boot?*

- It is very easy to develop Spring Based applications with Java or Groovy.
- It reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, * Spring Security etc.
- It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
- It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command * prompt very easily and quickly.
- It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
- It provides lots of plugins to work with embedded and in-memory Databases very easily.

↥ back to top

Q. *Write a program in Spring-Boot to get employee details based on employee id?*

We make use of the h2 database. Maven will be as follows-

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.springexample</groupId>
        <artifactId>SpringBootHelloWorld</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>

        <name>SpringBootHelloWorld</name>
        <description>Demo project for Spring Boot</description>

        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.4.1.RELEASE</version>
                <relativePath /> <!-- lookup parent from repository -->
        </parent>

        <properties>
                <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
                <project.reporting.outputEncoding>UTF-8</project.reporting.outputEnc
                <java.version>1.8</java.version>
        </properties>

        <dependencies>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>
                <dependency>
                        <groupId>com.h2database</groupId>
                        <artifactId>h2</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.apache.tomcat.embed</groupId>
                        <artifactId>tomcat-embed-jasper</artifactId>
                </dependency>
                <dependency>
                        <groupId>javax.servlet</groupId>
                        <artifactId>jstl</artifactId>
                </dependency>
```

```
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

Create the SpringBootHelloWorldApplication.java as follows-

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RestController;

@RestController
@EnableAutoConfiguration
@SpringBootApplication
public class SpringBootHelloWorldApplication {

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

Create the Entity class as follows-

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Employee {
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Id
	private long id;
```

```
        private String name;

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String getDept() {
                return dept;
        }

        public void setDept(String dept) {
                this.dept = dept;
        }

        private String dept;

        @Override
        public String toString() {
                return "Employee [id=" + id + ", name=" + name + ", dept=" + dept +
        }
}
```

The Controller we define methods to add Employee record and display employee records as list. Define the controller as follows-

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import com.springexample.data.EmployeeRepository;
import com.springexample.model.Employee;

@Controller
public class EmployeeController {

        @Autowired
        private EmployeeRepository employeeData;
```

```
        @RequestMapping(value = "/addNewEmployee.html", method = RequestMethod.POST)
        public String newEmployee(Employee employee) {
                employeeData.save(employee);
                return ("redirect:/list.html");
        }

        @RequestMapping(value = "/addNewEmployee.html", method = RequestMethod.GET)
        public ModelAndView addNewEmployee() {
                Employee emp = new Employee();
                return new ModelAndView("newEmployee", "form", emp);
        }

        @RequestMapping(value = "/listEmployees.html", method = RequestMethod.GET)
        public ModelAndView employees() {
                List<Employee> allEmployees = employeeData.findAll();
                return new ModelAndView("allEmployees", "employees", allEmployees);
        }
}
```

Define the newEmployee.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<h1>Employees page</h1>

<ul>
<c:forEach items="" var="employee">
   <li></li>
</c:forEach>
</ul>
```

Define the allEmployees.jsp

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
    <h1>Add new employee</h1>

    <form:form modelAttribute="form">
        <form:errors path="" element="div" />
        <div>
            <form:label path="name">Name</form:label>
            <form:input path="name" />
            <form:errors path="name" />
```

```
        </div>
        <div>
            <input type="submit" />
        </div>
    </form:form>
</body>
</html>
```

The application.properties will be as follows-

```
spring.mvc.view.prefix:/WEB-INF/jsp/
spring.mvc.view.suffix:.jsp

spring.datasource.url=jdbc:h2:file:./DB
spring.jpa.properties.hibernate.hbm2ddl.auto=update
```

↥ back to top

Q. *What does the @RestController, @RequestMapping, @RequestParam, @ContextConfiguration, @ResponseBody, @pathVariable, @ResponseEntity, @Qualifier annotation do?*

- **@RestController**: The @RestController is a stereotype annotation. It adds `@Controller` and `@ResponseBody` annotations to the class. It requires to import `org.springframework.web.bind.annotation` package. The @RestController annotation informs to the Spring to render the result back to the caller.

```
import org.springframework.web.bind.annotation.RestController;
@RestController  // using @RestController annotation
public class HomeController {
    // controller body
}
```

- **@RequestMapping**: The @RequestMapping annotation is used to provide routing information. It tells to the Spring that any HTTP request should map to the corresponding method. It requires to import `org.springframework.web.annotation` package. Example: Here method index() should map with `/index` url

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class HomeController {
    @RequestMapping(value = "/index", method = "GET")
    public String index() {
        return "Dashboard Page!";
    }
}
```

- **@RequestParam**: @RequestParam is a Spring annotation used to bind a web request parameter to a method parameter. It has the following optional elements:
    - **defaultValue**: used as a fallback when the request parameter is not provided or has an empty value
    - **name**: name of the request parameter to bind to
    - **required**: tells whether the parameter is required
    - **value**: alias for name

1. A Simple Mapping

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam String id) {
    return "ID: " + id;
}
```

Output

```
http://localhost:8080/api/foos?id=abc
----
ID: abc
```

2. Specifying the Request Parameter Name

```
@PostMapping("/api/foos")
@ResponseBody
public String addFoo(@RequestParam(name = "id") String fooId, @RequestParam String n
    return "ID: " + fooId + " Name: " + name;
}
```

3. Making an Optional Request Parameter

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam(required = false) String id) {
    return "ID: " + id;
}
```

Output

```
http://localhost:8080/api/foos?id=abc
----
ID: abc


http://localhost:8080/api/foos
----
ID: null
```

4. A Default Value for the Request Parameter

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam(defaultValue = "test") String id) {
    return "ID: " + id;
}
```

Output

```
http://localhost:8080/api/foos
----
ID: test


http://localhost:8080/api/foos?id=abc
----
ID: abc
```

5. Mapping All Parameters

```
@PostMapping("/api/foos")
@ResponseBody
public String updateFoos(@RequestParam Map<String,String> allParams) {
    return "Parameters are " + allParams.entrySet();
}
```

Output

```
curl -X POST -F 'name=abc' -F 'id=123' http://localhost:8080/api/foos
-----
Parameters are {[name=abc], [id=123]}
```

6. Mapping a Multi-Value Parameter

```
@GetMapping("/api/foos")
@ResponseBody
public String getFoos(@RequestParam List<String> id) {
    return "IDs are " + id;
}
```

Output

```
http://localhost:8080/api/foos?id=1,2,3
----
IDs are [1,2,3]

http://localhost:8080/api/foos?id=1&id=2
----
IDs are [1,2]
```

- **@ContextConfiguration**: This annotation specifies how to load the application context while writing a unit test for the Spring environment. Here is an example of using @ContextConfiguration along with @RunWith annotation of JUnit to test a Service class in Spring Boot.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=PaymentConfiguration.class)
public class PaymentServiceTests {

  @Autowired
  private PaymentService paymentService;

  @Test
  public void testPaymentService() {
      // code to test PaymentService class
  }
}
```

Here, `@ContextConfiguration` class instructs to load the Spring application context defined in the PaymentConfiguration class.

- **@ResponseBody**: The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

```
@Controller
@RequestMapping("/post")
public class ExamplePostController {

    @Autowired
    ExampleService exampleService;

    @PostMapping("/response")
    @ResponseBody
    public ResponseTransfer postResponseController(
      @RequestBody LoginForm loginForm) {
        return new ResponseTransfer("Thanks For Posting!!!");
     }
}
```

Output

```
{"text":"Thanks For Posting!!!"}
```

- **@pathVariable**: @PathVariable is a Spring annotation which indicates that a method parameter should be bound to a URI template variable. It has the following optional elements:
  - **name**: name of the path variable to bind to
  - **required**: tells whether the path variable is required
  - **value**: alias for name

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyController {

    @RequestMapping(path="/{name}/{age}")
    public String getMessage(@PathVariable("name") String name,
```

```
            @PathVariable("age") String age) {

        var msg = String.format("%s is %s years old", name, age);
        return msg;
    }
}
```

- **@ResponseEntity**: ResponseEntity represents an HTTP response, including headers, body, and status. While `@ResponseBody` puts the return value into the body of the response, ResponseEntity also allows us to add headers and status code.

```
@GetMapping("/customHeader")
ResponseEntity<String> customHeader() {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Custom-Header", "foo");

    return new ResponseEntity<>(
      "Custom header set", headers, HttpStatus.OK);
}
```

- **@Qualifier**: Spring Boot `@Qualifier` shows how to differentiate beans of the same type with @Qualifier. It can also be used to annotate other custom annotations that can then be used as qualifiers.

```
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
@Qualifier("manager")
public class Manager implements Person {

    @Override
    public String info() {
        return "Manager";
    }
}
```

↥ back to top

Q. *What are the different components of a Spring Boot application?*

Spring Boot Framework has mainly four major components.

- **Spring Boot Starters**: The main responsibility of Spring Boot Starter is to combine a group of common or related dependencies into single dependencies. Spring Boot starters can help to reduce the number of manually added dependencies just by adding one dependency. So instead of manually specifying the dependencies just add one starter. Examples are spring-boot-starter-web, spring-boot-starter-test, spring-boot-starter-data-jpa, etc.

- **Spring Boot AutoConfigurator**: One of the common complaint with Spring is, we need to make lot of XML based configurations. Spring Boot AutoConfigurator will simplify all these XML based configurations. It also reduces the number of annotations.

- **Spring Boot CLI**: Spring Boot CLI(Command Line Interface) is a Spring Boot software to run and test Spring Boot applications from command prompt. When we run Spring Boot applications using CLI, then it internally uses Spring Boot Starter and Spring Boot AutoConfigurate components to resolve all dependencies and execute the application.

- **Spring Boot Actuator**: Spring Boot Actuator is a sub-project of Spring Boot. It adds several production grade services to your application with little effort on your part. Actuators enable production-ready features to a Spring Boot application, without having to actually implement these things yourself. The Spring Boot Actuator is mainly used to get the internals of running application like health, metrics, info, dump, environment, etc. which is similar to your production environment monitoring setup.

↥ back to top

Q. *What does @SpringBootApplication and @EnableAutoConfiguration do?*

- **@SpringBootApplication**: annotation is used to annotate the main class of our Spring Boot application. It also enables the auto-configuration feature of Spring Boot.

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

- **@EnableAutoConfiguration**: The auto-configuration feature automatically configures things if certain classes are present in the Classpath. For example, if you have a data

source bean present in the classpath of the application, then it automatically configures the JDBC template.

```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class SpringBootDemo {
  //.. Java code
}
```

↥ back to top

Q. *What is Spring Boot initializr?*

The Spring Initializr is ultimately a web application that can generate a Spring Boot project structure for you. It doesn't generate any application code, but it will give you a basic project structure and either a Maven or a Gradle build specification to build your code with. All you need to do is write the application code.

Spring Initializr can be used several ways, including:

1. A web-based interface.
2. Via Spring ToolSuite.
3. Using the Spring Boot CLI.

↥ back to top

Q. *What is a profile? How do you create application configuration for a specific profile?*

Spring Profiles helps to segregating application configurations, and make them available only in certain environments. Any `@Component` or `@Configuration` can be marked with `@Profile` to limit when it is loaded. You can define default configuration in application.properties. Environment specific overrides can be configured in specific files:

- application-dev.properties
- application-qa.properties
- application-stage.properties
- application-prod.properties

Using Profiles In Code

```
@Configuration
@Profile("dev")
public class DevConfigurations {
    // DEV Configurations
}
@Configuration
@Profile("prod")
public class ProdConfigurations {
    // Production Configurations
}
```

↥ back to top

Q. *What is Spring Boot Actuator? How do you monitor web services using Spring Boot Actuator?*

Spring Boot Actuator module use to monitor and manage Spring Boot application by providing production-ready features like health check-up, auditing, metrics gathering, HTTP tracing etc. All of these features can be accessed over JMX or HTTP endpoints.

Adding Spring Boot Actuator

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

Monitoring: Actuator creates several so-called **endpoints** that can be exposed over HTTP or JMX to let you monitor and interact with application.

For example, There is a `/health` endpoint that provides basic information about the application's health. The `/metrics` endpoint shows several useful metrics information like JVM memory used, system CPU usage, open files, and much more. The `/loggers` endpoint shows application's logs and also lets you change the log level at runtime.

```
http://localhost:8080/actuator

----
{
  "_links": {
```

```
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    }
  }
}
```

Endpoint	Description
health	Application health info
info	Info about the application
env	Properties from environment
metrics	Various metrics about the app
mappings	@RequestMapping Controller mappings
shutdown	Triggers application shutdown
httptrace	HTTP request/response log
loggers	Display and configure logger info
logfile	Contents of the log file
threaddump	Perform thread dump
heapdump	Obtain JVM heap dump
caches	Check available caches
integrationgraph	Graph of Spring Integration components

Enabling / Disabling endpoints

```
# Disable an endpoint
management.endpoint.[endpoint-name].enabled=false

# Specific example for 'health' endpoint
```

```
management.endpoint.health.enabled=false

# Instead of enabled by default, you can change to mode
# where endpoints need to be explicitly enabled
management.endpoints.enabled-by-default=false
```

↥ back to top

Q. *What is a CommandLineRunner and ApplicationRunner?*

`ApplicationRunner` and `CommandLineRunner` interfaces use to execute the code after the Spring Boot application is started. These interfaces can be used to perform any actions immediately after the application has started.

- **CommandLineRunner** This interface provides access to application arguments as string array.

```
@Component
public class CommandLineAppStartupRunner implements CommandLineRunner {
    private static final Logger logger = LoggerFactory.getLogger(CommandLineAppStart
    @Override
    public void run(String...args) throws Exception {
        logger.info("Application started with command-line arguments: {} .
        \n To kill this application, press Ctrl + C.", Arrays.toString(args));
    }
}
```

- **ApplicationRunner** ApplicationRunner wraps the raw application arguments and exposes the ApplicationArguments interface, which has many convinent methods to get arguments, like getOptionNames() to return all the arguments' names, getOptionValues() to return the agrument value, and raw source arguments with method getSourceArgs().

```
@Component
public class AppStartupRunner implements ApplicationRunner {
    private static final Logger logger = LoggerFactory.getLogger(AppStartupRunner.cl
    @Override
    public void run(ApplicationArguments args) throws Exception {
        logger.info("Your application started with option names : {}", args.getOptic
    }
}
```

↥ back to top

Q. *What is Docker? How to deploy Spring Boot Application to Docker?*

A Docker is a tool that makes it very easy to deploy and run an application using **containers**. A container allows a developer to create an all-in-one package of the developed application with all its dependencies. For example, a Java application requires Java libraries, and when we deploy it on any system or VM, we need to install Java first. But, in a container, everything is kept together and shipped as one package, such as in a Docker container.

Step 01: **Create a simple Spring Boot Application**

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello Docker World";
    }

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

To run the application, use the following Maven command from the project root folder:

```
cmd> mvn spring-boot:run
```

Step 02: **Dockerizing using Dockerfile**

- **Dockerfile –** Specifying a file that contains native Docker commands to build the image
- **Maven –** Using a Maven plugin to build the image

A Dockerfile is just a regular `.txt` file that includes native Docker commands that are used to specify the layers of an image. The content of the file itself can look something like this:

```
FROM java:8-jdk-alpine

COPY ./target/demo-docker-0.0.1-SNAPSHOT.jar /usr/app/

WORKDIR /usr/app

RUN sh -c 'touch demo-docker-0.0.1-SNAPSHOT.jar'

ENTRYPOINT ["java","-jar","demo-docker-0.0.1-SNAPSHOT.jar"]
```

- **FROM –** The keyword FROM tells Docker to use a given base image as a build base. We have used 'java' with tag '8-jdk-alpine'. Think of a tag as a version. The base image changes from project to project. You can search for images on docker-hub.
- **COPY -** This tells Docker to copy files from the local file-system to a specific folder inside the build image. Here, we copy our .jar file to the build image (Linux image) inside /usr/app.
- **WORKDIR -** The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow in the Dockerfile. Here we switched the workdir to /usr/app so as we don't have to write the long path again and again.
- **RUN -** This tells Docker to execute a shell command-line within the target system. Here we practically just "touch" our file so that it has its modification time updated (Docker creates all container files in an "unmodified" state by default).
- **ENTRYPOINT -** This allows you to configure a container that will run as an executable. It's where you tell Docker how to run your application. We know we run our spring-boot app as java -jar .jar, so we put it in an array.

Step 03: **Create Docker image**

Generate a Spring Boot `.jar` file using `mvn clean install` command. This file will be used to create the Docker image. Let's build the image using this Dockerfile. To do so, move to the root directory of the application and run this command:

```
cmd> docker build -t greeting-app
```

We built the image using `docker build`. We gave it a name with the `-t` flag and specified the current directory where the Dockerfile is. The image is built and stored in our local docker registry.

Let's check our image:

```
cmd> docker images
```

And finally, let's run our image:

```
cmd> docker run -p 8090:8080 greeting-app
```

We can run Docker images using the `docker run` command.

Each container is an isolated environment in itself and we have to map the port of the host operating system - 8090 and the port inside the container - 8080, which is specified as the -p 8090:8080 argument. Now, we can access the endpoint on
`http://localhost:8080/greet/Pradeep`

↥ back to top

Q. *How to implement Exception Handling in Spring Boot?*

Spring Boot provides a number of options for error/exception handling.
1. @ExceptionHandler Annotation: This annotation works at the `@Controller` class level. The issue with the approach is only active for the given controller. The annotation is not global, so we need to implement in each and every controller.

```
@RestController
public class WelcomeController {

    @GetMapping("/greeting")
    String greeting() throws Exception {
      //
    }

    @ExceptionHandler({Exception.class})
    public  handleException(){
       //
    }
}
```

2. @ControllerAdvice Annotation: This annotation supports global Exception handler mechanism. So we can implement the controller exception handling events in a central location.

```
@ControllerAdvice
public class GlobalRestExceptionHandler extends ResponseEntityExceptionHandler {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(Exception.class)
    public void defaultExceptionHandler() {
        // Nothing to do
    }
}
```

3. @ResponseEntityExceptionHandler: This method can be used with `@ControllerAdvice` classes. It allows the developer to specify some specific templates of ResponseEntity and return values.

4. @RestControllerAdvice: Spring Boot 1.4 introduced the `@RestControllerAdvice` annotation for easier exception handling. It is a convenience annotation that is itself annotated with `@ControllerAdvice` and `@ResponseBody`.

```
@RestControllerAdvice
public class RestExceptionHandler {

@ExceptionHandler(CustomNotFoundException.class)
public ApiErrorResponse handleNotFoundException(CustomNotFoundException ex) {

ApiErrorResponse response = new ApiErrorResponse.ApiErrorResponseBuilder()
      .withStatus(HttpStatus.NOT_FOUND)
      .withError_code("NOT_FOUND")
      .withMessage(ex.getLocalizedMessage()).build();

    return responseMsg;
    }
}
```

↥ back to top

Q. *What is caching? Have you used any caching framework with Spring Boot?*

Caching is a mechanism to enhance the performance of a system. It is a temporary memory that lies between the application and the persistent database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Types of cache

- **In-memory caching**: This is the most frequently used area where caching is used extensively to increase performance of the application. In-memory caches such as `Memcached` and `Radis` are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk.
- **Database caching**: Database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance. One popular in this area is first level cache of `Hibernate` or any `ORM frameworks`.
- **Web server caching**: Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers. In today's API age, this option is a viable if we want to cache API responses in web server level.
- **CDN caching**: Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.

Spring Boot Cache Annotations

- **@EnableCaching**: It can be added to the boot application class annotated with `@SpringBootApplication`. Spring provides one concurrent hashmap as default cache, but we can override CacheManager to register external cache providers as well easily.
- **@Cacheable**: It is used on the method level to let spring know that the response of the method are cacheable. Spring manages the request/response of this method to the cache specified in annotation attribute.

```
@Cacheable(value="books", key="#isbn")
public Book findStoryBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

- **@CachePut**: It allow us to update the cache and will also allow the method to be executed. It supports the same options as `@Cacheable` and should be used for cache population rather then method flow optimization.
- **@CacheEvict**: It is used when we need to evict (remove) the cache previously loaded of master data. When CacheEvict annotated methods will be executed, it will clear the cache.

- **@Caching**: This annotation is required when we need both `@CachePut` and `@CacheEvict` at the same time.

Spring Boot Caching Example

- **Create HTTP GET REST API**

Student.java

```
public class Student {

    String id;
    String name;
    String clz;

    public Student(String id, String name, String clz) {
        super();
        this.id = id;
        this.name = name;
        this.clz = clz;
    }
    //Setters and getters
}
```

StudentService.java

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
import com.example.springcache.domain.Student;

@Service
public class StudentService {

    @Cacheable("student")
    public Student getStudentByID(String id) {

        try {
            System.out.println("Going to sleep for 5 Secs.. to simulate backend call
            Thread.sleep(1000*5);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        return new Student(id, "Pradeep", "V");
    }
}
```

StudentController.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import com.example.springcache.domain.Student;
import com.example.springcache.service.StudentService;

@RestController
public class StudentController {

    @Autowired
    StudentService studentService;

    @GetMapping("/student/{id}")
    public Student findStudentById(@PathVariable String id) {

        System.out.println("Searching by ID  : " + id);
        return studentService.getStudentByID(id);
    }
}
```

Enable Spring managed Caching

SpringCacheApplication.java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@EnableCaching
public class SpringCacheApplication {

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

Output

```
Searching by ID  : 1
Going to sleep for 5 Secs.. to simulate backend call.
```

```
Searching by ID  : 1
Searching by ID  : 1
Searching by ID  : 1
Searching by ID  : 1
Searching by ID  : 1

Searching by ID  : 2
Going to sleep for 5 Secs.. to simulate backend call.

Searching by ID  : 2
Searching by ID  : 2
```

↥ back to top

Q. *What is Swagger? Have you implemented it using Spring Boot?*

Swagger is widely used for visualizing APIs, and with Swagger UI it provides online sandbox for frontend developers. Swagger is a tool, a specification and a complete framework implementation for producing the visual representation of RESTful Web Services. It enables documentation to be updated at the same pace as the server. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

Create REST APIs

- Open `application.properties` and add below property. This will start the application in /swagger2-demo context path.

```
server.contextPath=/swagger2-demo
```

- Add one REST controller `Swagger2DemoRestController` which will provide basic REST based functionalities on Student entity.

Swagger2DemoRestController.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import com.example.springbootswagger2.model.Student;

@RestController
public class Swagger2DemoRestController {

    List<Student> students = new ArrayList<Student>();
    {
        students.add(new Student("Sajal", "IV", "India"));
        students.add(new Student("Lokesh", "V", "India"));
        students.add(new Student("Kajal", "III", "USA"));
        students.add(new Student("Sukesh", "VI", "USA"));
    }

    @RequestMapping(value = "/getStudents")
    public List<Student> getStudents() {
        return students;
    }

    @RequestMapping(value = "/getStudent/{name}")
    public Student getStudent(@PathVariable(value = "name") String name) {
        return students.stream().filter(x -> x.getName().equalsIgnoreCase(name)).col
    }

    @RequestMapping(value = "/getStudentByCountry/{country}")
    public List<Student> getStudentByCountry(@PathVariable(value = "country") String
        System.out.println("Searching Student in country : " + country);
        List<Student> studentsByCountry = students.stream().filter(x -> x.getCountry
                .collect(Collectors.toList());
        System.out.println(studentsByCountry);
        return studentsByCountry;
    }

    @RequestMapping(value = "/getStudentByClass/{cls}")
    public List<Student> getStudentByClass(@PathVariable(value = "cls") String cls)
        return students.stream().filter(x -> x.getCls().equalsIgnoreCase(cls)).colle
    }
}
```

Student.java

```
public class Student {

    private String name;
    private String cls;
    private String country;

    public Student(String name, String cls, String country) {
```

```
        super();
        this.name = name;
        this.cls = cls;
        this.country = country;
    }

    public String getName() {
        return name;
    }

    public String getCls() {
        return cls;
    }

    public String getCountry() {
        return country;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", cls=" + cls + ", country=" + country + "
    }
}
```

- Start the application as Spring boot application. Test couple of REST Endpoints to check if they are working fine:

```
http://localhost:8080/swagger2-demo/getStudents
http://localhost:8080/swagger2-demo/getStudent/sajal
http://localhost:8080/swagger2-demo/getStudentByCountry/india
http://localhost:8080/swagger2-demo/getStudentByClass/v
```

Swagger2 Configuration

- **Add Swagger2 Maven Dependencies**: Open pom.xml file of the spring-boot-swagger2 project and add below two swagger related dependencies i.e. springfox-swagger2 and springfox-swagger-ui.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>

<dependency>
```

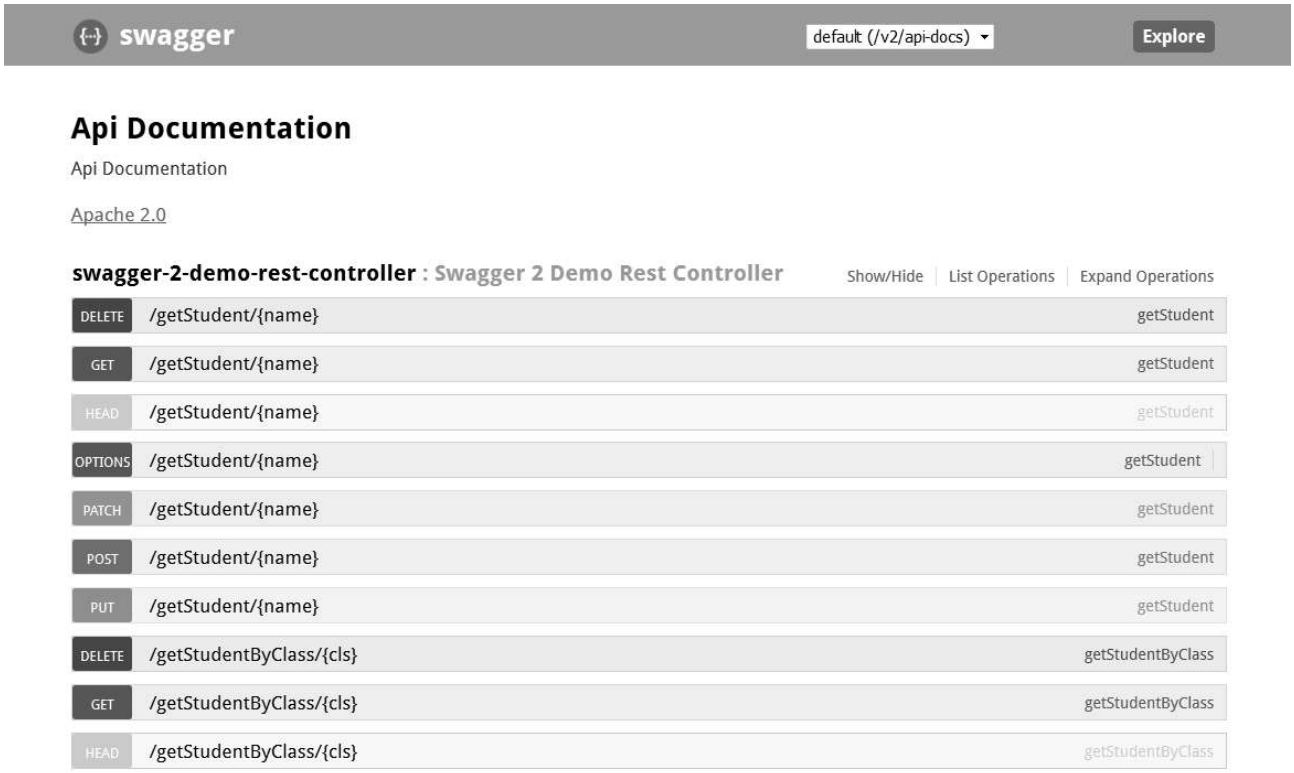
```
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
</dependency>
```

- **Add Swagger2 Configuration**: Add the below configuration in the code base. To help you understand the configuration, I have added inline comments.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import com.google.common.base.Predicates;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class Swagger2UiConfiguration extends WebMvcConfigurerAdapter
{
    @Bean
    public Docket api() {
        // @formatter:off
        //Register the controllers to swagger
        //Also it is configuring the Swagger Docket
        return new Docket(DocumentationType.SWAGGER_2).select()
                // .apis(RequestHandlerSelectors.any())
                .apis(Predicates.not(RequestHandlerSelectors.basePackage("org.spring
                // .paths(PathSelectors.any())
                // .paths(PathSelectors.ant("/swagger2-demo"))
                .build();
        // @formatter:on
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
    {
        //enabling swagger-ui part for visual documentation
        registry.addResourceHandler("swagger-ui.html").addResourceLocations("classpa
        registry.addResourceHandler("/webjars/**").addResourceLocations("classpath:/
    }
}
```

- **Verify Swagger2 UI Docs**

```
http://localhost:8080/swagger2-demo/swagger-ui.html
```



↥ back to top

Q. *How to implement Pagination and Sorting in Spring Boot?*

- **JPA Entity**:

EmployeeEntity.java

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="TBL_EMPLOYEES")
public class EmployeeEntity {

    @Id
    @GeneratedValue
    private Long id;
```

```
    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email", nullable=false, length=200)
    private String email;

    //Setters and getters

    @Override
    public String toString() {
        return "EmployeeEntity [id=" + id + ", firstName=" + firstName +
                ", lastName=" + lastName + ", email=" + email   + "]";
    }
}
```

- **PagingAndSortingRepository**
  PagingAndSortingRepository is an extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction. It provides two methods :
  - **Page findAll(Pageable pageable) –** returns a Page of entities meeting the paging restriction provided in the Pageable object.
  - **Iterable findAll(Sort sort) –** returns all entities sorted by the given options. No paging is applied here.

EmployeeRepository.java

```
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.stereotype.Repository;
import com.springbatchexample.demo.entity.EmployeeEntity;

@Repository
public interface EmployeeRepository
        extends PagingAndSortingRepository<EmployeeEntity, Long> {

}
```

- **Accepting paging and sorting parameters**
  In below spring mvc controller, we are accepting paging and sorting parameters using pageNo, pageSize and sortBy query parameters. Also, by default '10' employees will

be fetched from database in page number '0', and employee records will be sorted based on 'id' field.

EmployeeController.java

```
@RestController
@RequestMapping("/employees")
public class EmployeeController
{
    @Autowired
    EmployeeService service;

    @GetMapping
    public ResponseEntity<List<EmployeeEntity>> getAllEmployees(
                        @RequestParam(defaultValue = "0") Integer pageNo,
                        @RequestParam(defaultValue = "10") Integer pageSize,
                        @RequestParam(defaultValue = "id") String sortBy)
    {
        List<EmployeeEntity> list = service.getAllEmployees(pageNo, pageSize, sortBy

        return new ResponseEntity<List<EmployeeEntity>>(list, new HttpHeaders(), Htt
    }
}
```

To perform pagination and/or sorting, we must create org.springframework.data.domain.Pageable or org.springframework.data.domain.Sort instances are pass to the findAll() method.

EmployeeService.java

```
@Service
public class EmployeeService
{
    @Autowired
    EmployeeRepository repository;

    public List<EmployeeEntity> getAllEmployees(Integer pageNo, Integer pageSize, St
    {
        Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by(sortBy));

        Page<EmployeeEntity> pagedResult = repository.findAll(paging);

        if(pagedResult.hasContent()) {
            return pagedResult.getContent();
        } else {
```

```
            return new ArrayList<EmployeeEntity>();
        }
    }
}
```

- **Pagination and sorting techniques**
  - **Paging WITHOUT sorting**: To apply only pagination in result set, we shall create Pageable object without any Sort information.

```
Pageable paging = PageRequest.of(pageNo, pageSize);
Page<EmployeeEntity> pagedResult = repository.findAll(paging);
```

- **Paging WITH sorting**: To apply only pagination in result set, we shall create Pageable object with desired Sort column name.

```
Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by("email"));
Page<EmployeeEntity> pagedResult = repository.findAll(paging);
```

- **Sorting only**: If there is no need to page, and only sorting is required, we can create Sort object for that.

```
Sort sortOrder = Sort.by("email");
List<EmployeeEntity> list = repository.findAll(sortOrder);
```

↥ back to top

Q. *How to use schedulers in Spring Boot?*

Spring Boot internally uses the `TaskScheduler` interface for scheduling the annotated methods for execution. The @Scheduled annotation is added to a method along with some information about when to execute it, and Spring Boot takes care of the rest. **Enable Scheduling**

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.scheduling.annotation.EnableScheduling;

@SpringBootApplication
@EnableScheduling
public class SchedulerDemoApplication {
```

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

Scheduling a Task with Fixed Rate

```
@Scheduled(fixedRate = 2000)
public void scheduleTaskWithFixedRate() {
    logger.info("Fixed Rate Task :: Execution Time - {}", dateTimeFormatter.format(L
}
```

Sample Output

```
Fixed Rate Task :: Execution Time - 10:26:58
Fixed Rate Task :: Execution Time - 10:27:00
Fixed Rate Task :: Execution Time - 10:27:02
....
....
```

Scheduling a Task using Cron Expression

```
@Scheduled(cron = "0 * * * * ?")
public void scheduleTaskWithCronExpression() {
    logger.info("Cron Task :: Execution Time - {}", dateTimeFormatter.format(LocalDa
}
```

Sample Output

```
Cron Task :: Execution Time - 11:03:00
Cron Task :: Execution Time - 11:04:00
Cron Task :: Execution Time - 11:05:00
```

↥ back to top

Q. *How to provide security to spring boot application?*

pom.xml

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

create a MVC configuration file that extends WebMvcConfigurerAdapter.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {
   @Override
   public void addViewControllers(ViewControllerRegistry registry) {
      registry.addViewController("/home").setViewName("home");
      registry.addViewController("/").setViewName("home");
      registry.addViewController("/hello").setViewName("hello");
      registry.addViewController("/login").setViewName("login");
   }
}
```

create a Web Security Configuration file, that is used to secure your application to access the HTTP Endpoints by using basic authentication.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.Authen
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityC
import org.springframework.security.config.annotation.web.configuration.EnableWebSec

@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();
    }
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
       auth
          .inMemoryAuthentication()
          .withUser("user").password("password").roles("USER");
    }
}
```

↥ back to top

Q. *What is CORS in Spring Boot? How to enable CORS in Spring Boot?*

Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.

- **Enable CORS in Controller Method**

```
@RequestMapping(value = "/products")
@CrossOrigin(origins = "http://localhost:8080")
public ResponseEntity<Object> getProduct() {
   return null;
}
```

- **Global CORS Configuration** We need to define the shown `@Bean` configuration to set the CORS configuration support globally to your Spring Boot application.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
   return new WebMvcConfigurerAdapter() {
      @Override
      public void addCorsMappings(CorsRegistry registry) {
         registry.addMapping("/products").allowedOrigins("http://localhost:9000");
      }
   };
}
```

To code to set the CORS configuration globally in main Spring Boot application is given below.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@SpringBootApplication
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
   @Bean
   public WebMvcConfigurer corsConfigurer() {
      return new WebMvcConfigurerAdapter() {
         @Override
         public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/products").allowedOrigins("http://localhost:8080")
         }
      };
   }
}
```

↥ back to top

Q. *What is CSRF attack? How to enable CSRF protection against it?*

CSRF: CSRF stands for Cross-Site Request Forgery. It is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

In order to use the Spring Security CSRF protection, we'll first need to make sure we use the proper HTTP methods for anything that modifies state (PATCH, POST, PUT, and DELETE – not GET).

1. Java Configuration
CSRF protection is **enabled by default** in the Java configuration. We can still disable it if we need to:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
      .csrf().disable();
}
```

2. XML Configuration

Starting from Spring Security 4.x – the CSRF protection is enabled by default in the XML configuration as well; we can of course still disable it if we need to:

```
<http>
    ...
    <csrf disabled="true"/>
</http>
```

3. Extra Form Parameters

With CSRF protection enabled on the server side, we'll need to include the CSRF token in our requests on the client side as well:

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

4. Using JSON

We can't submit the CSRF token as a parameter if we're using JSON; instead we can submit the token within the header. We'll first need to include the token in our page – and for that we can use meta tags:

```
<meta name="_csrf" content="${_csrf.token}"/>
<meta name="_csrf_header" content="${_csrf.headerName}"/>
```

Then we'll construct the header:

```
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");

$(document).ajaxSend(function(e, xhr, options) {
    xhr.setRequestHeader(header, token);
});
```

↥ back to top

Q. *How do you configure error logging/debugging in Spring Boot application?*

In Spring Boot, Logback is the default logging framework, just add spring-boot-starter-web, it will pull in the logback dependencies.
pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-boot-slf4j</artifactId>
    <packaging>jar</packaging>
    <name>Spring Boot SLF4j</name>
    <version>1.0</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.2.RELEASE</version>
    </parent>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <!-- Package as an executable jar/war -->
```

```
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.0</version>
            </plugin>
        </plugins>
    </build>
</project>
```

- **application.properties**

```
# logging level
logging.level.org.springframework=ERROR
logging.level.com.mkyong=DEBUG

# output to a file
logging.file=app.log

# temp folder example
#logging.file=${java.io.tmpdir}/app.log

logging.pattern.file=%d %p %c{1.} [%t] %m%n

logging.pattern.console=%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n

## if no active profile, default is 'default'
##spring.profiles.active=prod

# root level
#logging.level.=INFO
```

- **logback.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <property name="HOME_LOG" value="logs/app.log"/>

    <appender name="FILE-ROLLING" class="ch.qos.logback.core.rolling.RollingFileAppe
        <file>${HOME_LOG}</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPol
```

```
            <fileNamePattern>logs/archived/app.%d{yyyy-MM-dd}.%i.log</fileNamePatter
            <!-- each archived file, size max 10MB -->
            <maxFileSize>10MB</maxFileSize>
            <!-- total size of all archive files, if total size > 20GB,
                                it will delete old archived file -->
            <totalSizeCap>20GB</totalSizeCap>
            <!-- 60 days to keep -->
            <maxHistory>60</maxHistory>
        </rollingPolicy>

        <encoder>
            <pattern>%d %p %c{1.} [%t] %m%n</pattern>
        </encoder>
    </appender>

    <logger name="com.mkyong" level="debug" additivity="false">
        <appender-ref ref="FILE-ROLLING"/>
    </logger>

    <root level="error">
        <appender-ref ref="FILE-ROLLING"/>
    </root>

</configuration>
```

HelloController.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import java.util.Arrays;
import java.util.List;

@Controller
public class HelloController {

    private static final Logger logger = LoggerFactory.getLogger(HelloController.cla

    @GetMapping("/")
    public String hello(Model model) {

        List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);

        logger.debug("Hello from Logback {}", data);
        model.addAttribute("num", data);
```

```
        return "index"; // index.html
    }
}
```

↥ back to top

Q. *What is Spring Batch? How do you implement it using Spring Boot?*

Spring Batch is a lightweight, comprehensive batch framework that is designed for use in developing robust batch applications.

Why Is Spring Batch Useful

- Restartability
- Different readers and writers
- Chunk Processing
- Ease Of Transaction Management
- Ease of parallel processing

Project Structure

In this project, we will create a simple job with 2 step tasks and execute the job to observe the logs. Job execution flow will be –

1. Start job
2. Execute task one
3. Execute task two
4. Finish job

- **Maven Dependencies**

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xs
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.springbatchexample</groupId>
    <artifactId>App</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

```
    <packaging>jar</packaging>

    <name>App</name>
    <url>http://maven.apache.org</url>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-batch</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>repository.spring.release</id>
            <name>Spring GA Repository</name>
            <url>http://repo.spring.io/release</url>
        </repository>
    </repositories>
</project>
```

- Add Tasklets

TaskOne.java

```
import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

public class TaskOne implements Tasklet {

    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkCon
    {
        System.out.println("TaskOne start..");
        // ... some code
        System.out.println("TaskOne done..");
        return RepeatStatus.FINISHED;
    }
}
```

TaskTwo.java

```
import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

public class TaskTwo implements Tasklet {

    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkCon
    {
        System.out.println("TaskTwo start..");
        // ... some code
        System.out.println("TaskTwo done..");
        return RepeatStatus.FINISHED;
    }
}
```

- **Spring Batch Configuration**
  This is major step where you define all the job related configurations and it's execution logic.

BatchConfig.java

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.springbatchexample.demo.tasks.TaskOne;
import com.springbatchexample.demo.tasks.TaskTwo;

@Configuration
@EnableBatchProcessing
public class BatchConfig {

    @Autowired
    private JobBuilderFactory jobs;

    @Autowired
    private StepBuilderFactory steps;

    @Bean
    public Step stepOne(){
        return steps.get("stepOne")
                .tasklet(new TaskOne())
                .build();
    }

    @Bean
    public Step stepTwo() {
        return steps.get("stepTwo")
                .tasklet(new TaskTwo())
                .build();
    }

    @Bean
    public Job demoJob() {
        return jobs.get("demoJob")
                .incrementer(new RunIdIncrementer())
                .start(stepOne())
                .next(stepTwo())
                .build();
    }
}
```

- **Demo**
  Now our simple job 'demoJob' is configured and ready to be executed. I am using CommandLineRunner interface to execute the job automatically, with JobLauncher, when the application is fully started.

App.java

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App implements CommandLineRunner {
    @Autowired
    JobLauncher jobLauncher;

    @Autowired
    Job job;

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

    @Override
    public void run(String... args) throws Exception {
        JobParameters params = new JobParametersBuilder()
                    .addString("JobID", String.valueOf(System.currentTimeMillis()))
                    .toJobParameters();
        jobLauncher.run(job, params);
    }
}
```

Console Logs

```
o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob: [name=demoJob]]
launched with
the following parameters: [{JobID=1530697766768}]

o.s.batch.core.job.SimpleStepHandler     : Executing step: [stepOne]
TaskOne start..
```

```
TaskOne done..

o.s.batch.core.job.SimpleStepHandler     : Executing step: [stepTwo]
TaskTwo start..
TaskTwo done..

o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob: [name=demoJob]]
completed with
the following parameters: [{JobID=1530697766768}] and the following status:
[COMPLETED]
```

↥ back to top

Q. *How to implement interceptor with Spring Boot?*

Interceptor can be used to perform operations in the following situations –

- Before sending the request to the controller
- Before sending the response to the client

For example, interceptor can be used to add the request header before sending the request to the controller and add the response header before sending the response to the client.

Interceptors support three methods –

- **preHandle()** – This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.
- **postHandle()** – This is used to perform operations before sending the response to the client.
- **afterCompletion()** – This is used to perform operations after completing the request and response.

```
@Component
public class ProductServiceInterceptor implements HandlerInterceptor {
   @Override
   public boolean preHandle(
      HttpServletRequest request, HttpServletResponse response, Object handler) thro

       log.info("[preHandle][" + request + "]" + "[" + request.getMethod()
      + "]" + request.getRequestURI() + getParameters(request));
      return true;
   }
   @Override
```

```
    public void postHandle(
       HttpServletRequest request, HttpServletResponse response, Object handler,
       ModelAndView modelAndView) throws Exception {
           log.info("[postHandle][" + request + "]");
       }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse respo
       Object handler, Exception ex) throws Exception {
            if (ex != null) {
               ex.printStackTrace();
            }
            log.info("[afterCompletion][" + request + "][exception: " + ex + "]");
       }
}
```

↥ back to top

Q. *How to use Form Login Authentication using Spring Boot?*

Include spring security 5 dependencies

pom.xml

```
<properties>
        <failOnMissingWebXml>false</failOnMissingWebXml>
        <spring.version>5.0.7.RELEASE</spring.version>
</properties>

<!-- Spring MVC Dependency -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>

<!-- Spring Security Core -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>${spring.version}</version>
</dependency>

<!-- Spring Security Config -->
```

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${spring.version}</version>
</dependency>

<!-- Spring Security Web -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${spring.version}</version>
</dependency>
```

- **Configure Authentication and URL Security**

SecurityConfig.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.authentication.builders.Authen
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSec
import org.springframework.security.config.annotation.web.configuration.WebSecurityC
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    PasswordEncoder passwordEncoder;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
        .passwordEncoder(passwordEncoder)
        .withUser("user").password(passwordEncoder.encode("123456")).roles("USER")
        .and()
        .withUser("admin").password(passwordEncoder.encode("123456")).roles("USER",
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
```

```
            http.authorizeRequests()
            .antMatchers("/login")
                .permitAll()
            .antMatchers("/**")
                .hasAnyRole("ADMIN", "USER")
            .and()
                .formLogin()
                .loginPage("/login")
                .defaultSuccessUrl("/home")
                .failureUrl("/login?error=true")
                .permitAll()
            .and()
                .logout()
                .logoutSuccessUrl("/login?logout=true")
                .invalidateHttpSession(true)
                .permitAll()
            .and()
                .csrf()
                .disable();
        }
    }
```

- Bind spring security to web application

SpringSecurityInitializer.java

```
import org.springframework.security.web.context.AbstractSecurityWebApplicationInitia
public class SpringSecurityInitializer extends AbstractSecurityWebApplicationInitial
    //no code needed
}
```

AppInitializer.java

```
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherSer

public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitial

   @Override
   protected Class<?>[] getRootConfigClasses() {
      return new Class[] { HibernateConfig.class, SecurityConfig.class };
   }

   @Override
   protected Class<?>[] getServletConfigClasses() {
```

```
        return new Class[] { WebMvcConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

- **Login Controller**

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.logout.SecurityContextLogoutH
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class LoginController
{
    @RequestMapping(value = "/login", method = RequestMethod.GET)
    public String loginPage(@RequestParam(value = "error", required = false) String
                            @RequestParam(value = "logout", required = false) String
                            Model model) {
        String errorMessge = null;
        if(error != null) {
            errorMessge = "Username or Password is incorrect !!";
        }
        if(logout != null) {
            errorMessge = "You have been successfully logged out !!";
        }
        model.addAttribute("errorMessge", errorMessge);
        return "login";
    }

    @RequestMapping(value="/logout", method = RequestMethod.GET)
    public String logoutPage (HttpServletRequest request, HttpServletResponse respon
        Authentication auth = SecurityContextHolder.getContext().getAuthentication()
        if (auth != null){
            new SecurityContextLogoutHandler().logout(request, response, auth);
        }
```

```
        return "redirect:/login?logout=true";
    }
}
```

login.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<body onload='document.loginForm.username.focus();'>
    <h1>Spring Security 5 - Login Form</h1>

    <c:if test="${not empty errorMessge}"><div style="color:red; font-weight: bold;

    <form name='login' action="/login" method='POST'>
        <table>
            <tr>
                <td>UserName:</td>
                <td><input type='text' name='username' value=''></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><input type='password' name='password' /></td>
            </tr>
            <tr>
                <td colspan='2'><input name="submit" type="submit" value="submit" />
            </tr>
        </table>
        <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
    </form>
</body>
</html>
```

Output

```
// Run
--------

http://localhost:8080/login
```

↥ back to top

Q. *What are the Spring Boot starters and what are available the starters?*

Spring Boot starters are a set of convenient dependency management providers which can be used in the application to enable dependencies. These starters, make development easy and rapid. All the available starters come under the `org.springframework.boot` group. Few of the popular starters are as follows:

- **spring-boot-starter**: This is the core starter and includes logging, auto-configuration support, and YAML.
- **spring-boot-starter-jdbc**: This starter is used for HikariCP connection pool with JDBC
- **spring-boot-starter-web**: Is the starter for building web applications, including RESTful, applications using Spring MVC
- **spring-boot-starter-data-jpa**: Is the starter to use Spring Data JPA with Hibernate
- **spring-boot-starter-security**: Is the starter used for Spring Security
- **spring-boot-starter-aop**: This starter is used for aspect-oriented programming with AspectJ and Spring AOP
- **spring-boot-starter-test**: Is the starter for testing Spring Boot applications

↥ back to top

Q. *What is GZIP? How to implement it using Spring Boot? How to enable HTTP response compression in Spring Boot?*

GZip compression is a very simple and effective way to save bandwidth and improve the speed of website. It reduces the response time of website by compressing the resources and then sending it over to the clients. It saves bandwidth by at least 50%.

GZip compression is disabled by default in Spring Boot. To enable it, add the following properties to your `application.properties` file

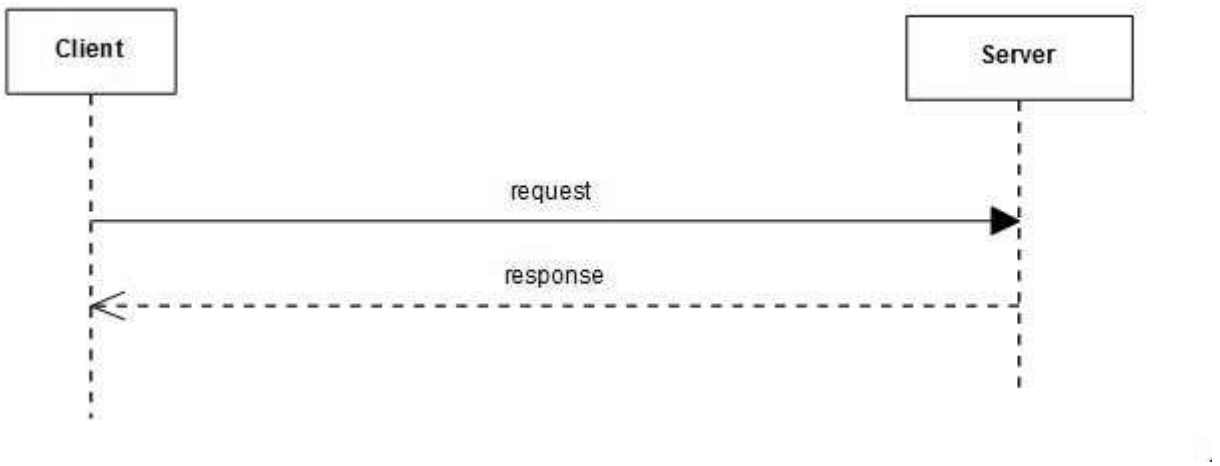
```
# Enable response compression
server.compression.enabled=true

# The comma-separated list of mime types that should be compressed
server.compression.mime-
types=text/html,text/xml,text/plain,text/css,text/javascript,application/json

# Compress the response only if the response size is at least 1KB
server.compression.min-response-size=1024
```

↥ back to top

Q. *When will you use WebSockets? How to implement it using Spring Boot?*

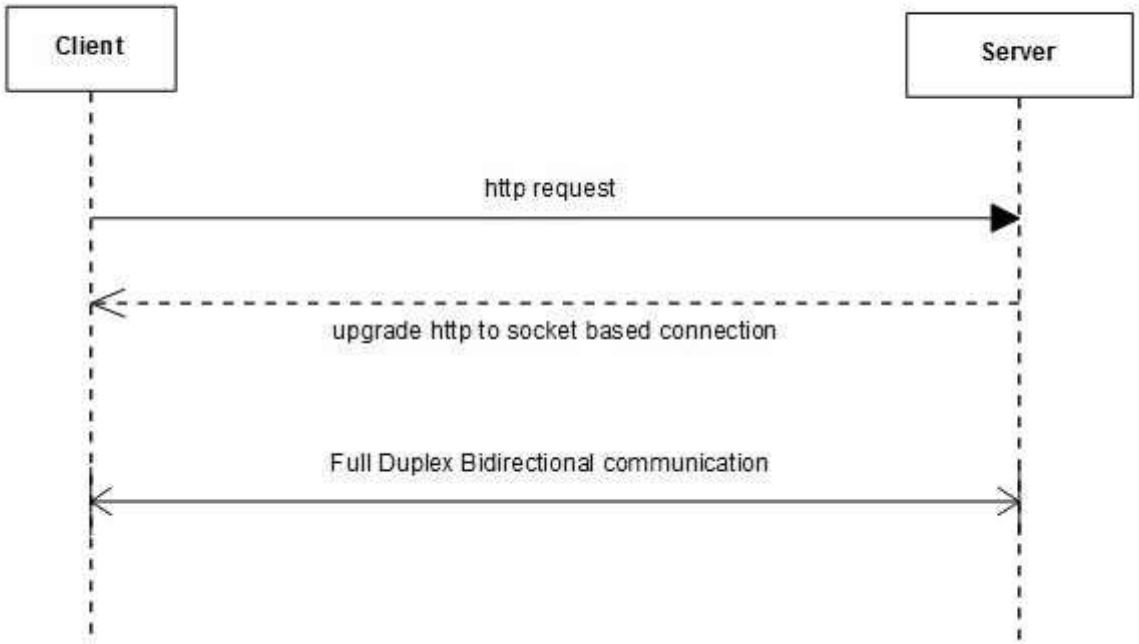
WebSocket is a protocol which enables communication between the server and the browser. It has an advantage over RESTful HTTP because communications are both bi-directional and real-time. This allows for the server to notify the client at any time instead of the client polling on a regular interval for updates.

Following are some of the drawbacks of HTTP due to which they are unsuitable for certain scenarios-



- **Traditional HTTP** requests are unidirectional - In traditional client server communication, the client always initiates the * request.
- **Half Duplex -** User requests for a resource and the server then serves it to the client. The response is only sent after the request. So at a time only a single request occurs.
- **Multiple TCP connections -** For each request a new TCPsession is needed to be established and then closed after receiving the response. So without using WebSockets we will have multiple sessions.
- **Heavy -** Normal HTTP request and response require exchange of extra data between client and server.

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection.



- **WebSocket are bi-directional** - Using WebSocket either client or server can initiate sending a message.
- **WebSocket are Full Duplex -** The client and server communication is independent of each other.
- **Single TCP connection -** The initial connection is using HTTP, then this connection gets upgraded to a socket based * connection. This single connection is then used for all the future communication
- **Light -** The WebSocket message data exchange is much lighter compared to http.

WebSockets Implementation

In the Maven we need the spring boot WebSocket dependency.Maven will be as follows-

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.javaexample</groupId>
        <artifactId>boot-websocket</artifactId>
        <version>1.0-SNAPSHOT</version>

        <parent>
                <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.4.1.RELEASE</version>
        </parent>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-websocket</artifactId>
                </dependency>

                <dependency>
                        <groupId>org.json</groupId>
                        <artifactId>json</artifactId>
                        <version>20171018</version>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

Create the SpringBoot Bootstrap class as below-

```
package com.javaexample.websocket.config;

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

@SpringBootApplication
public class Application {

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

On the Server end, we recieve the data and reply back to the client. In Spring we can create a customized handler by using either TextWebSocketHandler or BinaryWebSocketHandler.

```
package com.javaexample.websocket.config;
import java.io.IOException;
import org.json.JSONObject;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHandler;

@Component
public class SocketTextHandler extends TextWebSocketHandler {

        @Override
        public void handleTextMessage(WebSocketSession session, TextMessage message)
                        throws InterruptedException, IOException {

                String payload = message.getPayload();
                JSONObject jsonObject = new JSONObject(payload);
                session.sendMessage(new TextMessage("Hi " + jsonObject.get("user") +
        }

}
```

In order to tell Spring to forward client requests to the endpoint , we need to register the handler.

```
package com.javaexample.websocket.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

        public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
                registry.addHandler(new SocketTextHandler(), "/user");
        }

}
```

Next we define the UI part for establishing WebSocket and making the calls- Define the app.js as follows-

```
var ws;
function setConnected(connected) {
        $("#connect").prop("disabled", connected);
        $("#disconnect").prop("disabled", !connected);
}

function connect() {
        ws = new WebSocket('ws://localhost:8080/user');
        ws.onmessage = function(data) {
                helloWorld(data.data);
        }
        setConnected(true);
}

function disconnect() {
        if (ws != null) {
                ws.close();
        }
        setConnected(false);
        console.log("Websocket is in disconnected state");
}

function sendData() {
        var data = JSON.stringify({
                'user' : $("#user").val()
        })
        ws.send(data);
}

function helloWorld(message) {
        $("#helloworldmessage").append(" " + message + "");
}

$(function() {
        $("form").on('submit', function(e) {
                e.preventDefault();
        });
        $("#connect").click(function() {
                connect();
        });
        $("#disconnect").click(function() {
                disconnect();
        });
        $("#send").click(function() {
                sendData();
        });
});
```

Define the index.html as follows-

```
<!DOCTYPE html>
<html>
<head>
    <title>WebSocket Chat Application </title>
    <link href="/bootstrap.min.css" rel="stylesheet">
    <link href="/style.css" rel="stylesheet">
    <script src="/jquery-1.10.2.min.js"></script>
    <script src="/app.js"></script>
</head>
<body>
<div id="main-content" class="container">
    <div class="row">
        <div class="col-md-8">
            <form class="form-inline">
                <div class="form-group">
                    <label for="connect">Chat Application:</label>
                    <button id="connect" type="button">Start New Chat</button>
                    <button id="disconnect" type="button" disabled="disabled">End Ch
                    </button>
                </div>
            </form>
        </div>
    </div>
    <div class="row">
        <div class="col-md-12">
            <table id="chat">
                <thead>
                <tr>
                    <th>Welcome user. Please enter you name</th>
                </tr>
                </thead>
                <tbody id="helloworldmessage">
                </tbody>
            </table>
        </div>
            <div class="row">

        <div class="col-md-6">
            <form class="form-inline">
                <div class="form-group">
                    <textarea id="user" placeholder="Write your message here..." req
                </div>
                <button id="send" type="submit">Send</button>
            </form>
        </div>
        </div>
```

```
      </div>
    </div>
  </body>
  </html>
```

Start the application- http://localhost:8080 Click on start new chat it opens the WebSocket connection.

↥ back to top

Q. *What is Spring Boot devtools?*

The aim of this module is to try and improve the development-time experience when working on Spring Boot applications.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    </dependency>
</dependencies>
```

↥ back to top

Q. *What is the configuration file name, which is used by Spring Boot?*

```
application.properties
```

↥ back to top

Q. *What is difference Between an Embedded Container and a WAR?*

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. By default the embedded server will listen for HTTP requests on port 8080.

JAR

You can run independently every appliction with different ports (in linux, **java -jar ... > app_logs.log &**) and you can route it (e.g. nginx). Note that, restarting is not problem. You can write custom bash script (like this: **ps aux | grep appname** and kill by PID). But there are some problems with configuring production app. Property files will archived into jar.

WAR

You can deploy into container and just run it. Easy managing at the server. If you want to re-configure app, open properties file from unarchived folder inside container, change it as need and restart container. So, managing and configuring will be easy. But, if you want to run another app in this server with another port, then you must install another copy of container and config it.

↥ back to top

Q. *What is Mockito?*

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications. Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

Spring Boot - mockito and junit – unit test service layer Example

- **Maven Dependencies**
  The spring-boot-starter-test dependency includes all required dependencies to create and execute tests.

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

- **MockitoJUnitRunner class**: It automatically initialize all the objects annotated with `@Mock` and `@InjectMocks` annotations.

```
@RunWith(MockitoJUnitRunner.class)
public class TestEmployeeManager {

    @InjectMocks
    EmployeeManager manager;

    @Mock
    EmployeeDao dao;
```

```
    //tests
}
```

- **JUnit tests using Mockito**

Here `getAllEmployees()` which will return list of EmployeeVO objects, `getEmployeeById(int id)` to return a employee by given id; and `createEmployee()` which will add an employee object and return void.

- **Service layer tests** (TestEmployeeManager.java)

```
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.junit.MockitoJUnitRunner;

import com.javaexample.demo.dao.EmployeeDao;
import com.javaexample.demo.model.EmployeeVO;
import com.javaexample.demo.service.EmployeeManager;

public class TestEmployeeManager {

    @InjectMocks
    EmployeeManager manager;

    @Mock
    EmployeeDao dao;

    @Before
    public void init() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void getAllEmployeesTest()
```

```
        {
            List<EmployeeVO> list = new ArrayList<EmployeeVO>();
            EmployeeVO empOne = new EmployeeVO(1, "John", "John", "javaexample@gmail.com
            EmployeeVO empTwo = new EmployeeVO(2, "Alex", "kolenchiski", "alexk@yahoo.co
            EmployeeVO empThree = new EmployeeVO(3, "Steve", "Waugh", "swaugh@gmail.com"

            list.add(empOne);
            list.add(empTwo);
            list.add(empThree);

            when(dao.getEmployeeList()).thenReturn(list);

            //test
            List<EmployeeVO> empList = manager.getEmployeeList();

            assertEquals(3, empList.size());
            verify(dao, times(1)).getEmployeeList();
        }

        @Test
        public void getEmployeeByIdTest()
        {
            when(dao.getEmployeeById(1)).thenReturn(new EmployeeVO(1,"Lokesh","Gupta","u

            EmployeeVO emp = manager.getEmployeeById(1);

            assertEquals("Lokesh", emp.getFirstName());
            assertEquals("Gupta", emp.getLastName());
            assertEquals("user@email.com", emp.getEmail());
        }

        @Test
        public void createEmployeeTest()
        {
            EmployeeVO emp = new EmployeeVO(1,"Lokesh","Gupta","user@email.com");

            manager.addEmployee(emp);

            verify(dao, times(1)).addEmployee(emp);
        }
    }
```

Service layer class (EmployeeManager.java)

```
    import java.util.List;

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

```
import org.springframework.stereotype.Service;

import com.javaexample.demo.dao.EmployeeDao;
import com.javaexample.demo.model.EmployeeVO;

@Service
public class EmployeeManager
{
    @Autowired
    EmployeeDao dao;

    public List<EmployeeVO> getEmployeeList() {
        return dao.getEmployeeList();
    }

    public EmployeeVO getEmployeeById(int id) {
        return dao.getEmployeeById(id);
    }

    public void addEmployee(EmployeeVO employee) {
        dao.addEmployee(employee);
    }
}
```

Dao layer class (EmployeeDao.java)

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.stereotype.Repository;

import com.javaexample.demo.model.EmployeeVO;

@Repository
public class EmployeeDao {

    private Map<Integer, EmployeeVO> DB = new HashMap<>();

    public List<EmployeeVO> getEmployeeList()
    {
        List<EmployeeVO> list = new ArrayList<>();
        if(list.isEmpty()) {
            list.addAll(DB.values());
        }
        return list;
    }
```

```
    public EmployeeVO getEmployeeById(int id) {
        return DB.get(id);
    }

    public void addEmployee(EmployeeVO employee) {
        employee.setEmployeeId(DB.keySet().size() + 1);
        DB.put(employee.getEmployeeId(), employee);
    }

    public void updateEmployee(EmployeeVO employee) {
        DB.put(employee.getEmployeeId(), employee);
    }

    public void deleteEmployee(int id) {
        DB.remove(id);
    }
}
```

↥ back to top

Q. *What is @SpringBootTest?*

- **@SpringBootTest for integration testing**

`@SpringBootTest` tries to mimic the processes added by Spring Boot framework for creating the context e.g. it decides what to scan based on package structures, loads external configurations from predefined locations, optionally runs auto-configuration starters and so on.

```
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class SpringBootDemoApplicationTests
{
    @LocalServerPort
    int randomServerPort;

    //---- tests -----
}
```

- **@SpringBootTest for unit testing**

`@SpringBootTest` annotation loads whole application, but it is better to limit Application Context only to a set of spring components that participate in test scenario.

The classes attribute specifies the annotated classes to use for loading an ApplicationContext.

```
@SpringBootTest(classes = {EmployeeRepository.class, EmployeeService.class})
public class SpringBootDemoApplicationTests
{
    @Autowired
    private EmployeeService employeeService;
    //---- tests -----
}
```

↥ back to top

Q. *How Spring boot autowiring an interface with multiple implementations?*

Use `@Qualifier` annotation is used to differentiate beans of the same interface

```
@SpringBootApplication
public class DemoApplication {

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

public interface MyService {
    void doWork();
}

@Service
@Qualifier("firstService")
public static class FirstServiceImpl implements MyService {

    @Override
    public void doWork() {
        System.out.println("firstService work");
    }
}

@Service
@Qualifier("secondService")
public static class SecondServiceImpl implements MyService {

    @Override
    public void doWork() {
```

```
            System.out.println("secondService work");
        }
    }

    @Component
    public static class FirstManager {

        private final MyService myService;

        @Autowired // inject FirstServiceImpl
        public FirstManager(@Qualifier("firstService") MyService myService) {
            this.myService = myService;
        }

        @PostConstruct
        public void startWork() {
            System.out.println("firstManager start work");
            myService.doWork();
        }
    }

    @Component
    public static class SecondManager {

        private final List<MyService> myServices;

        @Autowired // inject MyService all implementations
        public SecondManager(List<MyService> myServices) {
            this.myServices = myServices;
        }

        @PostConstruct
        public void startWork() {
            System.out.println("secondManager start work");
            myServices.forEach(MyService::doWork);
        }
      }
    }
```

↥ back to top

Q. ***What is the use of thymeleaf in spring boot?***

Q. ***What is difference between @Controller and @RestController in spring boot?***

Q. ***What is the use of servlet initializer in spring boot?***

↥ back to top

Releases

No releases published

Packages

No packages published

Contributors 2


learning-zone Pradeep Kumar


Ruthwik Ruthwik

Languages

- **Java** 100.0%