Spring Boot Assessment - 13

Section I [Multiple Choice Questions]

MCQ:

1. Which among these is not an application server provided by Spring Boot?
   Answer: D. Binary link
   (Spring Boot provides embedded application servers like Tomcat, Jetty, Undertow. "Binary link" is not a server.)

2. What is the starting point of a Spring Boot application?
   Answer: A. @SpringBootApplication
   (The annotation @SpringBootApplication is the primary starting point that enables auto-configuration and component scanning in a Spring Boot app.)

3. Which of the annotations is not a Spring boot annotation?
   Answer: D. @Data
   (@Data is from Lombok, not Spring Boot)

4. What is the Annotation used to handle GET requests?
   Answer: D. @GetMapping

5. What is the Annotation used for the Rest controller?
   Answer: C. @RestController

6. What is the prefix used in HTML for Thymeleaf?
   Answer: C. th:

7. How to bind Java variables with Thymeleaf?
   Answer: D. ${model.attribute}

8. How will you include a dependency in a Project?
   Answer: B. Add dependency inside pom.xml file

9. What is the dependency needed to create a Spring boot web Application?
   Answer: A. spring-boot-starter-web

10. Which among these does Spring Boot not provide?
    Answer: B. Equalizer

11. Database Objects must be annotated with
    Answer: B. @Entity

12. @Autowired can be used for
    Answer: D. All the Above
    (It can include Repository, Service, Component in a Controller or elsewhere)

13. What is the Right way to include a Repository into a Controller?
    Answer: B. @Autowired Repository myRepository

14. How to get All the Data from Customer Table using CustomerRepository?
    Answer: A. CustomerRepository.findAll()

15. Interceptors hooks
    Answer: E. All the Above
    (preHandle(), postHandle(), afterCompletion() are standard interceptor methods - assuming "beforeExecution()" is a distractor)

16. What is a microservice?
    Answer: C. A style of design for enterprise systems based on a loosely coupled component architecture

17. Which of the following responses is an advantage of microservices?
    Answer: A. Any microservice component can change independently from other components

18. What is a popular Java framework to develop microservices?
    Answer: C. Both A and B

19. How does Microservice architecture work?
    Answer: E. All the Above

20. Microservices can make use for betterment of app development
    Answer: D. All of the Above

Answers:

1) **D. Binary Link**
2) **A. @SpringBootApplication**
3) **D. @Data**
4) **D. @GetMapping**
5) **C. @RestController**
6) **C. th:**
7) **D. ${model.attribute}**

8) **B. Add dependency inside pom.xml file**

9) **A. spring-boot-starter-web**

10) **B. Equalizer**

11) **B. @Entity**

12) **D. All the Above (Repository, service, component in a controller)**

13) **B. @Autowired Repository myRepository**

14) **A. CustomerRepository.findAll()**

15) **E. All the Above (preHandle(), postHandle(), afterCompletion(), beforeExecution())**

16) **C. A style of design for enterprise systems based on a loosely coupled component architecture**

17) **A. Any microservice component can change independently from other components**

18) **C. Both A and B (Springboot , Eclipse Microprofile)**

19) **E. All the Above (An application is fragmented into loosely coupled various modules, each of which performs a distinct function. It is distributed across clouds and data centers. Each application module is an independent service/process that can be replaced, updated, or deleted without disrupting the rest of the application. Under microservice architecture, an application can grow along with its requirements)**

20) **D. All of the Above (RabbitMQ, Zuul and Hystri, Caching)**

Section II [Questions]

1. **What Is Spring Boot and Why is Spring Boot over Spring? List the features of Spring Boot that make it different?**

   Spring Boot is an extension of the Spring framework that makes it much easier and faster to create stand-alone, production-ready Spring applications. Here's the explanation of what Spring Boot is and why it's preferred over traditional Spring, along with its key features:

   What is Spring Boot?

- Spring Boot simplifies the process of building Spring applications by providing default configurations and removing most of the boilerplate setup required by regular Spring.
- It allows developers to create stand-alone applications that can run independently without the need to deploy to an external server through embedded servers like Tomcat or Jetty.
- Spring Boot is designed to accelerate development, especially for building microservices and RESTful services, with minimal upfront configuration.

Why Spring Boot over Spring?

- Less setup and configuration: Traditional Spring requires lots of manual XML or Java config. Spring Boot uses autoconfiguration to set sensible defaults, so you write less code.
- Embedded servers: Spring Boot includes embedded HTTP servers (Tomcat, Jetty), so you don't need to deploy WAR files to external servers - you can run your app as a simple jar.
- Opinionated defaults: Provides out-of-the-box defaults to get your application running quickly, but still allows customization.
- Simplified dependency management: Uses “starter” dependencies that aggregate common libraries, so you don’t have to manage versions manually.
- Production-ready features: Comes bundled with health checks, metrics, monitoring, and externalized configuration, making your app ready for production quickly.
- Rapid development and prototyping: Lower learning curve and faster time to market, helpful especially when building microservices architectures.

Key Features of Spring Boot:

1. Auto-configuration: Automatically configures components when it detects them on the classpath, eliminating manual setup.
2. Standalone: Runs as a self-contained jar with embedded server; no need for deploying to a separate web server.
3. Starters: Predefined dependency descriptors to simplify adding dependencies to your project.
4. Actuator: Provides built-in production-ready features like health checks, metrics, and monitoring endpoints.
5. Externalized Configuration: Supports property files, YAML, environment variables, and command-line arguments to configure your app separately from code.
6. No XML: Unlike traditional Spring, uses annotations and convention over configuration.
7. Developer tools: Supports hot swapping, auto-restart, and live reload with Spring Boot DevTools to improve developer productivity.
8. Wide Ecosystem Integration: Easily integrates with databases, messaging

systems, cloud services, security, etc.

9. Command-line interface: Spring Boot CLI allows running and testing Spring Boot apps from the terminal.

2. Illustrate a sample to Spring Boot Data Access.

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

```
package training.iqgateway.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import training.iqgateway.entities.BillingEO;
import training.iqgateway.service.BillingService;
@RestController
@RequestMapping("/billing")
@CrossOrigin(origins = "http://localhost:5173")
public class BillingController {
	@Autowired
	private BillingService billingService;
	@PostMapping
	public ResponseEntity<BillingEO> createOrUpdateBilling(@RequestBody
BillingEO billing) {
		BillingEO savedBilling = billingService.saveOrUpdateBilling(billing);
		return ResponseEntity.status(201).body(savedBilling);
	}
	@GetMapping("/{id}")
	public ResponseEntity<BillingEO> getBillingById(@PathVariable Integer id) {
```

```
                return
billingService.getBillingById(id).map(ResponseEntity::ok).orElse(ResponseEntity.notFou
nd().build());
        }
        @GetMapping
        public ResponseEntity<List<BillingEO>> getAllBillings() {
                List<BillingEO> billings = billingService.getAllBillings();
                return ResponseEntity.ok(billings);
        }
        @DeleteMapping("/{id}")
        public ResponseEntity<Void> deleteBilling(@PathVariable Integer id) {
                billingService.deleteBilling(id);
                return ResponseEntity.noContent().build();
        }
        @GetMapping("/appointment/{appointmentId}")
        public ResponseEntity<BillingEO> getBillingByAppointmentId(@PathVariable
Integer appointmentId) {
                return
billingService.getBillingByAppointmentId(appointmentId).map(ResponseEntity::ok)
                                .orElse(ResponseEntity.notFound().build());
        }

        @GetMapping("/patient/{patientId}")
        public ResponseEntity<List<BillingEO>> getBillingsByPatientId(@PathVariable
Integer patientId) {
           List<BillingEO> billings = billingService.getBillingsByPatientId(patientId);
           return ResponseEntity.ok(billings);
        }

        @GetMapping("/hospital/{hospitalId}")
        public ResponseEntity<List<BillingEO>> getBillingsByHospitalId(@PathVariable
Integer hospitalId) {
           List<BillingEO> billings = billingService.getBillingsByHospitalId(hospitalId);
           return ResponseEntity.ok(billings);
        }
}
```

package training.iqgateway.entities;

```
import java.util.Date;
import java.util.List;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import lombok.AllArgsConstructor;
```

```
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "billing")
public class BillingEO {
  @Id
  private Integer id;
  private Integer appointmentId;
  private List<String> billType;
  private Double amount;
  private String paymentMode;
  private String transactionId;
  private Date billingDate;
  private String billNo;
  private Integer patientId;
  private Integer hospitalId;
  private byte[] receipt;
}
package training.iqgateway.repository;
import java.util.List;
import java.util.Optional;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.stereotype.Repository;
import training.iqgateway.entities.BillingEO;
@Repository
public interface BillingRepository extends MongoRepository<BillingEO, Integer> {
  Optional<BillingEO> findByAppointmentId(Integer appointmentId);
  Optional<BillingEO> findTopByOrderByIdDesc();

  List<BillingEO> findByPatientId(Integer patientId);
        List<BillingEO> findByHospitalId(Integer hospitalId);
}


package training.iqgateway.service;
import java.util.List;
import java.util.Optional;
import training.iqgateway.entities.BillingEO;
public interface BillingService {
  BillingEO saveOrUpdateBilling(BillingEO billing);
  Optional<BillingEO> getBillingById(Integer id);
  List<BillingEO> getAllBillings();
```

```
  void deleteBilling(Integer id);
  Optional<BillingEO> getBillingByAppointmentId(Integer appointmentId);
  List<BillingEO> getBillingsByPatientId(Integer patientId);
        List<BillingEO> getBillingsByHospitalId(Integer hospitalId);
}
```

package training.iqgateway.service.impl;

```
import java.security.SecureRandom;
import java.util.List;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import training.iqgateway.entities.BillingEO;
import training.iqgateway.repository.BillingRepository;
import training.iqgateway.service.BillingService;
@Service
public class BillingServiceImpl implements BillingService {
        @Autowired
        private BillingRepository billingRepository;
        @Override
        public BillingEO saveOrUpdateBilling(BillingEO billing) {
                if (billing.getId() != null) {
                        Optional<BillingEO> existingBillingOpt =
billingRepository.findById(billing.getId());
                        if (existingBillingOpt.isPresent()) {
                                BillingEO existingBilling = existingBillingOpt.get();


existingBilling.setAppointmentId(billing.getAppointmentId());
                                existingBilling.setBillType(billing.getBillType());
                                existingBilling.setAmount(billing.getAmount());

existingBilling.setPaymentMode(billing.getPaymentMode());
                                existingBilling.setTransactionId(billing.getTransactionId());
                                existingBilling.setBillingDate(billing.getBillingDate());
                                existingBilling.setBillNo(billing.getBillNo());
                                existingBilling.setPatientId(billing.getPatientId());
                                existingBilling.setHospitalId(billing.getHospitalId());
                                if (billing.getReceipt() != null) {
                                        existingBilling.setReceipt(billing.getReceipt());
                                }
                                return billingRepository.save(existingBilling);
                        }
                }
                if (billing.getBillNo() == null || billing.getBillNo().isEmpty()) {
```

```
                billing.setBillNo(generateRandomBillNo());
            }
            Optional<BillingEO> maxBilling =
billingRepository.findTopByOrderByIdDesc();
            int newId = maxBilling.map(bill -> bill.getId() + 1).orElse(1);
            billing.setId(newId);
            return billingRepository.save(billing);
        }
        @Override
        public Optional<BillingEO> getBillingById(Integer id) {
            return billingRepository.findById(id);
        }
        @Override
        public List<BillingEO> getAllBillings() {
            return billingRepository.findAll();
        }
        @Override
        public void deleteBilling(Integer id) {
            billingRepository.deleteById(id);
        }
        @Override
        public Optional<BillingEO> getBillingByAppointmentId(Integer appointmentId) {
            return billingRepository.findByAppointmentId(appointmentId);
        }
        @Override
        public List<BillingEO> getBillingsByPatientId(Integer patientId) {
            return billingRepository.findByPatientId(patientId);
        }
        @Override
        public List<BillingEO> getBillingsByHospitalId(Integer hospitalId) {
            return billingRepository.findByHospitalId(hospitalId);
        }
}
```

3. Create a Spring Boot Application, demonstrating the usage of Themeleaf for Offence Management [Including Validation]

// Offence Entity

```
@Entity
public class Offence {
    @Id
    @GeneratedValue
```

```
    private Long id;
    @NotBlank(message = "Type is required")
    private String type;
        @Min(value = 1, message = "Severity must be at least 1")
    private int severity;
    // getters, setters
}



// Controller
@Controller
public class OffenceController {
    @Autowired
    private OffenceService offenceService;
    @GetMapping("/offences")
    public String showOffences(Model model) {
        model.addAttribute("offences", offenceService.getAllOffences());
        model.addAttribute("offence", new Offence());
        return "offences";
    }

    @PostMapping("/addOffence")
    public String addOffence(@Valid Offence offence, BindingResult result, Model model) {
        if (result.hasErrors()) {
            model.addAttribute("offences", offenceService.getAllOffences());
            return "offences";
        }
        offenceService.saveOffence(offence);
        return "redirect:/offences";
    }
}

// Thymeleaf Template (offences.html)
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Offence Management</title>
</head>
<body>
    <form th:action="@{/addOffence}" th:object="${offence}" method="post">
        <div>
            <label>Type:</label>
            <input type="text" th:field="*{type}"/>
            <span th:if="${#fields.hasErrors('type')}" th:errors="*{type}"></span>
```

```
        </div>
        <div>
            <label>Severity:</label>
            <input type="number" th:field="*{severity}"/>
            <span th:if="${#fields.hasErrors('severity')}" th:errors="*{severity}"></span>
        </div>
        <button type="submit">Add Offence</button>
    </form>

    <table>
        <tr th:each="offence : ${offences}">
            <td th:text="${offence.type}"></td>
            <td th:text="${offence.severity}"></td>
        </tr>
    </table>
</body>
</html>
```

4. Explain:

a. Interceptors and Internationalization in Spring Boot

Interceptors are used to intercept HTTP requests and responses. They can perform operations before and after controller execution.

```
public class LoggingInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response,
                             Object handler) {
        // Executed before controller
        System.out.println("Request URL: " + request.getRequestURL());
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response,
                           Object handler,
```

```
                    ModelAndView modelAndView) {
        // Executed after controller but before view rendering
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
                                HttpServletResponse response,
                                Object handler,
                                Exception ex) {
        // Executed after complete request completion
    }}
```

// Register interceptor

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoggingInterceptor());
    }}
```

Internationalization allows applications to support multiple languages:

1. Create messages.properties files:
    - messages.properties (default)
    - messages_fr.properties (French)
    - messages_es.properties (Spanish)
2. Configure in application.properties:
    - spring.messages.basename=messages

3. Use in Thymeleaf

```
<p th:text="#{welcome.message}"></p>
```

b. Validation Process in Spring Boot

Spring Boot uses Hibernate Validator for validation:

1. Annotate model with validation constraints:

```
public class User {
    @NotBlank
    @Size(min=3, max=30)
    private String name;
    @Email
    private String email;
    @Min(18)
    private int age;
}
```

2. Enable validation in controller:

```
@PostMapping("/users")
public String createUser(@Valid User user, BindingResult result) {
   if (result.hasErrors()) {
      return "userForm";
   }
   // process valid user
   return "success";
}
```

3. Show errors in Thymeleaf:

```
<input type="text" th:field="*{name}"/>
<span th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></span>
```

c. Spring Boot Actuator, Lombok and MapStruct

Spring Boot Actuator provides production-ready features:

- Add dependency: spring-boot-starter-actuator
- Endpoints: /health, /info, /metrics, /env, etc.
- Configure in application.properties:

```
management.endpoints.web.exposure.include=*

management.endpoint.health.show-details=always
```

Lombok reduces boilerplate code:

```
@Data // generates getters, setters, toString, equals, hashCode
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Product {
   @Id
   @GeneratedValue
   private Long id;
   private String name;
   private double price;
}
```

MapStruct simplifies mapping between DTOs and entities:

```
@Mapper(componentModel = "spring")
public interface ProductMapper {
   ProductDTO productToProductDTO(Product product);
   Product productDTOToProduct(ProductDTO productDTO);
}
// Usage:
```

```
@Autowired
private ProductMapper productMapper;
ProductDTO dto = productMapper.productToProductDTO(product);
```

5. Differences between Microservices and Monolithic Architecture

Monolithic Architecture:

- Single codebase for entire application
- Tightly coupled components
- Single database
- Scales by cloning entire application
- Single technology stack
- Deployed as single unit

Microservices Architecture:

- Multiple independent services
- Loosely coupled components
- Each service has its own database
- Scales individual services
- Polyglot persistence and technology
- Independently deployable

Sample Microservices Application:

1. Create three services:
    - User Service
    - Product Service
    - Order Service
2. Each with its own Spring Boot application and database
3. Communicate via REST APIs or messaging (RabbitMQ)
4. Example Order Service controller:

```
@RestController
@RequestMapping("/orders")
public class OrderController {
   @Autowired
   private UserServiceClient userServiceClient;
   @Autowired
   private ProductServiceClient productServiceClient;
   @PostMapping
   public Order createOrder(@RequestBody OrderRequest request) {
      User user = userServiceClient.getUser(request.getUserId());
```

```
        List<Product> products = productServiceClient.getProducts(request.getProductIds());
    return orderService.createOrder(user, products);    }}
```

6. Explain with suitable Illustrations

A. RabbitMQ

RabbitMQ is a message broker that implements AMQP protocol, used for asynchronous communication between microservices.

Configuration:

```
@Configuration
public class RabbitMQConfig {
   public static final String QUEUE = "orderQueue";
   @Bean
   Queue queue() {
      return new Queue(QUEUE, false);
   }}
```

Producer:

```
@Service
public class OrderProducer {
   @Autowired
   private RabbitTemplate rabbitTemplate;
   public void sendOrder(Order order) {
      rabbitTemplate.convertAndSend(RabbitMQConfig.QUEUE, order);
   }}
```

Consumer:

```
@Service
public class OrderConsumer {
   @RabbitListener(queues = RabbitMQConfig.QUEUE)
   public void receiveOrder(Order order) {
   }
}
```

B. Load Balancing: Spring Cloud LoadBalancer distributes traffic among service instances.

With Feign Client:

```
@FeignClient(name = "user-service")
public interface UserServiceClient {
   @GetMapping("/users/{id}")
   User getUser(@PathVariable Long id);
```

```
}
```

With RestTemplate:

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
   return new RestTemplate();
}

// Usage:
User user = restTemplate.getForObject("http://user-service/users/1", User.class);
```

C. Zuul and Hystrix

Zuul is an API gateway that provides:

- Dynamic routing
- Monitoring
- Security
- Load balancing

Configuration:

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

// application.properties

zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

D. Hystrix

Hystrix provides circuit breaker pattern:

```
@Service
public class UserService {
   @HystrixCommand(fallbackMethod = "getDefaultUser")
   public User getUser(Long id) {
      // call user service
```

```
    }
    public User getDefaultUser(Long id) {
        return new User(id, "Default", "User");
    }
}
```