Exercise 8: Online Bookstore - Implementing CRUD Operations

1. CRUD Endpoints for Book and Customer Entities:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

   @Autowired
   private BookService bookService;

   @PostMapping
   public ResponseEntity<Book> createBook(@Valid @RequestBody Book book) {
      return new ResponseEntity<>(bookService.saveBook(book), HttpStatus.CREATED);
   }

   @GetMapping("/{id}")
   public ResponseEntity<Book> getBookById(@PathVariable Long id) {
      return ResponseEntity.ok(bookService.getBookById(id));
   }

   @PutMapping("/{id}")
   public ResponseEntity<Book> updateBook(@PathVariable Long id, @Valid @RequestBody
Book bookDetails) {
      return ResponseEntity.ok(bookService.updateBook(id, bookDetails));
   }

   @DeleteMapping("/{id}")
   public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
      bookService.deleteBook(id);
      return ResponseEntity.noContent().build();
   }
}

@RestController
@RequestMapping("/api/customers")
public class CustomerController {

   @Autowired
   private CustomerService customerService;

   @PostMapping
```

```
    public ResponseEntity<Customer> createCustomer(@Valid @RequestBody Customer
customer) {
        return new ResponseEntity<>(customerService.saveCustomer(customer),
HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Customer> getCustomerById(@PathVariable Long id) {
        return ResponseEntity.ok(customerService.getCustomerById(id));
    }

    @PutMapping("/{id}")
    public ResponseEntity<Customer> updateCustomer(@PathVariable Long id, @Valid
@RequestBody Customer customerDetails) {
        return ResponseEntity.ok(customerService.updateCustomer(id, customerDetails));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteCustomer(@PathVariable Long id) {
        customerService.deleteCustomer(id);
        return ResponseEntity.noContent().build();
    }
}
```

2. **Validating Input Data:**

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Size(min = 1, max = 100)
    private String title;

    @NotNull
    @Size(min = 1, max = 100)
    private String author;

    @Min(0)
    private Double price;

    @Version
    private Integer version;
    // Getters and Setters
}

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Size(min = 1, max = 50)
    private String name;

    @NotNull
    @Size(min = 1, max = 100)
    private String email;

    @Version
    private Integer version;

    // Getters and Setters
}
```

- **@NotNull: Ensures that fields are not null.**
- **@Size: Validates the length of string fields.**
- **@Min: Ensures the price is not negative.**

3. Optimistic Locking:

Optimistic locking is managed using the `@Version` annotation, which ensures that during an update, the version number is checked. If it doesn't match, an `OptimisticLockException` is thrown, indicating a concurrent modification.

Exercise 9: Online Bookstore - Understanding HATEOAS

1. Adding Links to Resources using Spring HATEOAS:

First, include the necessary dependency in your `pom.xml`:

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

Next, modify your controllers to return HATEOAS-compliant responses:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

  @Autowired
  private BookService bookService;

  @GetMapping("/{id}")
  public ResponseEntity<EntityModel<Book>> getBookById(@PathVariable Long id) {
    Book book = bookService.getBookById(id);
    EntityModel<Book> resource = EntityModel.of(book);
    WebMvcLinkBuilder linkToBooks =
      linkTo(methodOn(this.getClass()).getAllBooks());
    resource.add(linkToBooks.withRel("all-books"));
    return ResponseEntity.ok(resource);
  }

  @GetMapping
  public ResponseEntity<CollectionModel<EntityModel<Book>>> getAllBooks() {
    List<EntityModel<Book>> books = bookService.getAllBooks().stream()
      .map(book -> EntityModel.of(book,
              linkTo(methodOn(this.getClass()).getBookById(book.getId())).withSelfRel()))
      .collect(Collectors.toList());
    return ResponseEntity.ok(CollectionModel.of(books));
  }
}

@RestController
@RequestMapping("/api/customers")
public class CustomerController {

  @Autowired
```

```
    private CustomerService customerService;

    @GetMapping("/{id}")
    public ResponseEntity<EntityModel<Customer>> getCustomerById(@PathVariable Long id) {
        Customer customer = customerService.getCustomerById(id);
        EntityModel<Customer> resource = EntityModel.of(customer);
        WebMvcLinkBuilder linkToCustomers =
            linkTo(methodOn(this.getClass()).getAllCustomers());
        resource.add(linkToCustomers.withRel("all-customers"));
        return ResponseEntity.ok(resource);
    }

    @GetMapping
    public ResponseEntity<CollectionModel<EntityModel<Customer>>> getAllCustomers() {
        List<EntityModel<Customer>> customers = customerService.getAllCustomers().stream()
            .map(customer -> EntityModel.of(customer,

linkTo(methodOn(this.getClass()).getCustomerById(customer.getId())).withSelfRel()))
            .collect(Collectors.toList());
        return ResponseEntity.ok(CollectionModel.of(customers));
    }
}
```

2. Hypermedia-Driven APIs:

- **Link Creation: The `EntityModel` class is used to create resource representations that include links to other resources.**
- **Hypermedia as the Engine of Application State (HATEOAS): This principle ensures that clients can navigate the API dynamically by following links, making the API more flexible and less dependent on hardcoded URLs.**

Exercise 10: Online Bookstore - Configuring Content Negotiation

1. Content Negotiation Configuration:

In Spring Boot, content negotiation can be configured using `ContentNegotiationConfigurer` or by defining the media types in the `application.properties` file.

Approach 1: Using `WebMvcConfigurer`

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.favorPathExtension(false)
                .favorParameter(true)
                .parameterName("mediaType")
                .ignoreAcceptHeader(false)
                .useRegisteredExtensionsOnly(false)
                .defaultContentType(MediaType.APPLICATION_JSON)
                .mediaType("json", MediaType.APPLICATION_JSON)
                .mediaType("xml", MediaType.APPLICATION_XML);
    }
}
```

Approach 2: Using `application.properties`

Add the following properties in your `application.properties` file:

```
spring.mvc.contentnegotiation.favor-path-extension=false
spring.mvc.contentnegotiation.favor-parameter=true
spring.mvc.contentnegotiation.parameter-name=mediaType
spring.mvc.contentnegotiation.ignore-accept-header=false
spring.mvc.contentnegotiation.default-content-type=application/json
spring.mvc.contentnegotiation.media-types.json=application/json
spring.mvc.contentnegotiation.media-types.xml=application/xml
```

2. Accept Header Implementation:

The REST controllers can automatically produce and consume different media types based on the `Accept` header due to the configuration done above.

Example REST Controller:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

   @Autowired
   private BookService bookService;

   @GetMapping(value = "/{id}", produces = { MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE })
   public ResponseEntity<Book> getBookById(@PathVariable Long id) {
      Book book = bookService.getBookById(id);
      return ResponseEntity.ok(book);
   }

   @PostMapping(consumes = { MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE },
             produces = { MediaType.APPLICATION_JSON_VALUE,
MediaType.APPLICATION_XML_VALUE })
   public ResponseEntity<Book> createBook(@Valid @RequestBody Book book) {
      Book createdBook = bookService.saveBook(book);
      return new ResponseEntity<>(createdBook, HttpStatus.CREATED);
   }
}
```

The `produces` attribute in the `@GetMapping` and `@PostMapping` annotations indicates that the endpoint can return either JSON or XML based on the `Accept` header.
The `consumes` attribute ensures that the API can accept input in both JSON and XML formats.

Exercise 11: Online Bookstore - Integrating Spring Boot Actuator

1. Adding Actuator Dependency:

Add the Spring Boot Actuator dependency to your `pom.xml` file:

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

2. Exposing and Customizing Actuator Endpoints:

By default, Spring Boot Actuator provides a variety of endpoints, such as `/health`, `/metrics`, `/info`, etc. You can customize which endpoints are exposed by configuring them in `application.properties`.

Example Configuration:

```
management.endpoints.web.exposure.include=health,info,metrics,env
management.endpoint.health.show-details=always
management.endpoint.metrics.enabled=true
```

- **`management.endpoints.web.exposure.include`: Specifies which Actuator endpoints should be exposed.**
- **`management.endpoint.health.show-details`: Determines whether detailed health information should be available.**
- **`management.endpoint.metrics.enabled`: Enables or disables the `/metrics` endpoint.**

3. Exposing Custom Metrics:

You can create custom metrics to monitor specific aspects of your application using `MeterRegistry`.

Example Custom Metric:

```
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CustomMetrics {
    @Autowired
    public CustomMetrics(MeterRegistry registry) {
        registry.gauge("custom.book.count", this, CustomMetrics::getBookCount);
    }

    public double getBookCount() {
        // Replace with logic to fetch the actual book count
        return 100;  // Example count
    }
}
```

- **The custom metric `custom.book.count` can now be monitored via the `/metrics` Actuator endpoint.**
- **Use the `MeterRegistry` to register custom metrics that track specific application data, such as the number of books in your store.**

Accessing Actuator Endpoints:

Once configured, you can access the Actuator endpoints by navigating to URLs like:

- `http://localhost:8080/actuator/health` **- Health check**
- `http://localhost:8080/actuator/metrics` **- Application metrics**
- `http://localhost:8080/actuator/custom.book.count` **- Custom metric**

Exercise 12: Online Bookstore - Securing RESTful Endpoints with Spring Security

1. Add Spring Security Dependency:

Add the Spring Security and JWT dependencies to your `pom.xml`:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt</artifactId>
   <version>0.9.1</version>
</dependency>
```

2. JWT Authentication Implementation:

Step 1: Create a JWT Utility Class

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Component
public class JwtUtil {

   private String secret = "mySecretKey";

   public String extractUsername(String token) {
      return extractClaim(token, Claims::getSubject);
   }

   public Date extractExpiration(String token) {
      return extractClaim(token, Claims::getExpiration);
   }
```

```
    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public String generateToken(String username) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, username);
    }

    private String createToken(Map<String, Object> claims, String subject) {
        return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new
Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
                .signWith(SignatureAlgorithm.HS256, secret).compact();
    }

    public Boolean validateToken(String token, String username) {
        final String extractedUsername = extractUsername(token);
        return (extractedUsername.equals(username) && !isTokenExpired(token));
    }
}
```

Step 2: Implement Security Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerB
uilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapt
er;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Autowired
   private JwtRequestFilter jwtRequestFilter;

   @Autowired
   private MyUserDetailsService myUserDetailsService;

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(myUserDetailsService).passwordEncoder(passwordEncoder());
   }

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

   @Override
   @Bean
   public AuthenticationManager authenticationManagerBean() throws Exception {
      return super.authenticationManagerBean();
   }

   @Override
   protected void configure(HttpSecurity httpSecurity) throws Exception {
      httpSecurity.csrf().disable()
            .authorizeRequests().antMatchers("/authenticate").permitAll()
            .anyRequest().authenticated()
            .and().sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
      httpSecurity.addFilterBefore(jwtRequestFilter,
UsernamePasswordAuthenticationFilter.class);
   }
}
```

Step 3: Implement JWT Request Filter

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import io.jsonwebtoken.ExpiredJwtException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    @Autowired
    private MyUserDetailsService myUserDetailsService;

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain chain)
            throws ServletException, IOException {

        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            try {
                username = jwtUtil.extractUsername(jwt);
            } catch (ExpiredJwtException e) {
                e.printStackTrace();
            }
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
```

```
      UserDetails userDetails = this.myUserDetailsService.loadUserByUsername(username);

      if (jwtUtil.validateToken(jwt, userDetails.getUsername())) {

        UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =
new UsernamePasswordAuthenticationToken(
            userDetails, null, userDetails.getAuthorities());
        usernamePasswordAuthenticationToken
            .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken)
;
      }
    }
    chain.doFilter(request, response);
  }
}
```

Step 4: Handle Authentication and Generate JWT

```
@RestController
public class AuthController {

  @Autowired
  private AuthenticationManager authenticationManager;

  @Autowired
  private JwtUtil jwtUtil;

  @Autowired
  private MyUserDetailsService userDetailsService;

  @PostMapping("/authenticate")
  public ResponseEntity<?> createAuthenticationToken(@RequestBody
AuthenticationRequest authenticationRequest) throws Exception {

    try {
      authenticationManager.authenticate(
          new
UsernamePasswordAuthenticationToken(authenticationRequest.getUsername(),
authenticationRequest.getPassword())
      );
    } catch (BadCredentialsException e) {
      throw new Exception("Incorrect username or password", e);
```

```
        }

        final UserDetails userDetails =
userDetailsService.loadUserByUsername(authenticationRequest.getUsername());
        final String jwt = jwtUtil.generateToken(userDetails.getUsername());

        return ResponseEntity.ok(new AuthenticationResponse(jwt));
    }
}
```

3. CORS Handling:

You can configure CORS in your `SecurityConfig` class:

```
@Override
protected void configure(HttpSecurity httpSecurity) throws Exception {
    httpSecurity.cors().and().csrf().disable()
            .authorizeRequests().antMatchers("/authenticate").permitAll()
            .anyRequest().authenticated()
            .and().sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
}

@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("http://localhost:3000"));
    configuration.setAllowedMethods(Arrays.asList("GET", POST, PUT, DELETE, OPTIONS"));
    configuration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"));
    configuration.setAllowCredentials(true);
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

Exercise 13: Online Bookstore - Unit Testing REST Controllers

1. JUnit Setup:

Add the necessary dependencies to your `pom.xml`:

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

2. Writing Unit Tests with MockMvc:

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;

@RunWith(SpringRunner.class)
@WebMvcTest(BookController.class)
public class BookControllerTest {

   @Autowired
   private MockMvc mockMvc;

   @MockBean
   private BookService bookService;

   @Test
   public void shouldReturnBookById() throws Exception {
      Book book = new Book(1L, "Spring in Action", "Craig Walls", 500.0);
      Mockito.when(bookService.getBookById(1L)).thenReturn(book);

      mockMvc.perform(get("/api/books/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.title").value("Spring in Action"))
            .andExpect(jsonPath("$.author").value("Craig Walls"));
   }
}
```

3. Test Coverage:

- **Ensure comprehensive test coverage by writing tests for all CRUD operations, edge cases, and exception handling scenarios.**

Exercise 14: Online Bookstore - Integration Testing for REST Services

1. Spring Test Setup:

Add the necessary testing dependencies to your `pom.xml`:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>test</scope>
</dependency>
```

Ensure your test class is annotated properly:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class BookstoreIntegrationTest {

   @Autowired
   private MockMvc mockMvc;

   @Autowired
   private BookRepository bookRepository;

   @Before
   public void setUp() {
      bookRepository.deleteAll();
   }

   // Integration test methods here
}
```

2. MockMvc Integration:

Create integration tests for your RESTful services:

```
@Test
public void whenPostRequestToBooks_thenCorrectResponse() throws Exception {
   String bookJson = "{\"title\": \"Spring in Action\", \"author\": \"Craig Walls\", \"price\": 500.0}";
```

```
    mockMvc.perform(post("/api/books")
            .content(bookJson)
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isCreated())
            .andExpect(jsonPath("$.title").value("Spring in Action"))
            .andExpect(jsonPath("$.author").value("Craig Walls"));
}

@Test
public void whenGetRequestToBooks_thenCorrectResponse() throws Exception {
    Book book = new Book(null, "Spring in Action", "Craig Walls", 500.0);
    bookRepository.save(book);

    mockMvc.perform(get("/api/books"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$[0].title").value("Spring in Action"))
            .andExpect(jsonPath("$[0].author").value("Craig Walls"));
}
```

3. Database Integration:

Configure H2 in-memory database for testing:

```
# application-test.properties
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=create-drop
```

Make sure to use the `@ActiveProfiles("test")` annotation in your test classes to load this configuration.

Exercise 15: Online Bookstore - API Documentation with Swagger

1. Add Swagger Dependency:

Add the Swagger (Springdoc) dependency to your `pom.xml`:

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.6.11</version>
</dependency>
```

2. Document Endpoints:

Annotate your REST controllers and methods:

```
@RestController
@RequestMapping("/api/books")
public class BookController {

   @Operation(summary = "Get all books")
   @GetMapping
   public List<Book> getAllBooks() {
      return bookService.findAllBooks();
   }

   @Operation(summary = "Add a new book")
   @PostMapping
   public ResponseEntity<Book> addBook(@RequestBody Book book) {
      return new ResponseEntity<>(bookService.saveBook(book), HttpStatus.CREATED);
   }
}
```

3. API Documentation:

Start your application and access the Swagger UI at `http://localhost:8080/swagger-ui.html` or Springdoc UI at `http://localhost:8080/swagger-ui/index.html`.