Exercise 7: Data Transfer Objects (DTOs)

1. **BookDTO and CustomerDTO Classes:**

```
// BookDTO.java
public class BookDTO {
    private Long id;
    private String title;
    private String author;
    private double price;

    // Getters and Setters
}

// CustomerDTO.java
public class CustomerDTO {
    private Long id;
    private String name;
    private String email;
    // Getters and Setters
}
```

2. **Mapping Entities to DTOs:**

Using **MapStruct**:

```
// BookMapper.java
@Mapper(componentModel = "spring")
public interface BookMapper {
    BookDTO toBookDTO(Book book);
    Book toBook(BookDTO bookDTO);
}

// CustomerMapper.java
@Mapper(componentModel = "spring")
public interface CustomerMapper {
```

```
    CustomerDTO toCustomerDTO(Customer customer);
    Customer toCustomer(CustomerDTO customerDTO);
}
```

Using **ModelMapper**:

```
// Configuration.java
@Bean
public ModelMapper modelMapper() {
    return new ModelMapper();
}
// Example usage:
BookDTO bookDTO = modelMapper.map(book, BookDTO.class);
Book book = modelMapper.map(bookDTO, Book.class);
```

3. **Custom Serialization/Deserialization:**

```
// Customizing JSON Serialization
@JsonInclude(JsonInclude.Include.NON_NULL)
public class BookDTO {
    @JsonProperty("book_id")
    private Long id;
    @JsonProperty("book_title")
    private String title;
    // Other fields and annotations
}
```

Exercise 8: CRUD Operations

1. **CRUD Endpoints:**

```
// BookController.java
@RestController
@RequestMapping("/api/books")
public class BookController {
    @Autowired
    private BookService bookService;
```

```
    @PostMapping
    public ResponseEntity<BookDTO> createBook(@Valid @RequestBody BookDTO bookDTO) {
        return ResponseEntity.ok(bookService.createBook(bookDTO));
    }
    @GetMapping("/{id}")
    public ResponseEntity<BookDTO> getBook(@PathVariable Long id) {
        return ResponseEntity.ok(bookService.getBook(id));
    }
    @PutMapping("/{id}")
    public ResponseEntity<BookDTO> updateBook(@PathVariable Long id, @Valid @RequestBody
BookDTO bookDTO) {
        return ResponseEntity.ok(bookService.updateBook(id, bookDTO));
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
        bookService.deleteBook(id);
        return ResponseEntity.noContent().build();
    }
}
// Similar for CustomerController.java
```

2. **Validating Input Data:**

```
public class BookDTO {
    @NotNull(message = "Title is required")
    @Size(min = 2, message = "Title should have at least 2 characters")
    private String title;
    @NotNull(message = "Author is required")
    @Size(min = 2, message = "Author should have at least 2 characters")
    private String author;
    @Min(value = 0, message = "Price must be a positive value")
    private double price;
```

```
    // Other fields and annotations
}
```

3. **Optimistic Locking:**

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Version
    private int version;
    // Other fields and methods
}
```

Exercise 9: HATEOAS

1. **Add Links to Resources:**

```
// BookController.java
@GetMapping("/{id}")
public EntityModel<BookDTO> getBook(@PathVariable Long id) {
    BookDTO bookDTO = bookService.getBook(id);
    EntityModel<BookDTO> resource = EntityModel.of(bookDTO);
    WebMvcLinkBuilder linkToBooks = linkTo(methodOn(this.getClass()).getAllBooks());
    resource.add(linkToBooks.withRel("all-books"));
    return resource;
}
// Similar for other methods
```

2. **Hypermedia-Driven APIs:**

- This is achieved by adding HATEOAS links as shown above.

Exercise 10: Content Negotiation

1. **Content Negotiation Configuration:**

```
// WebConfig.java
@Configuration
public class WebConfig implements WebMvcConfigurer {
```

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

2. **Accept Header:**

- This is handled by Spring's built-in content negotiation based on the Accept header.

Exercise 11: Spring Boot Actuator

1. **Add Actuator Dependency:**

```
<!-- Add to pom.xml -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. **Expose Actuator Endpoints:**

```
# application.properties
management.endpoints.web.exposure.include=health,info,metrics
```

3. **Custom Metrics:**

```
// CustomMetrics.java
@Component
public class CustomMetrics {

    private final MeterRegistry meterRegistry;
```

```
    @Autowired
    public CustomMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        registerCustomMetrics();
    }

    private void registerCustomMetrics() {
        Gauge.builder("custom.metric.book.count", this, CustomMetrics::getBookCount)
            .description("Number of books available")
            .register(meterRegistry);
    }

    public int getBookCount() {
        // Return the number of books
    }
}
```

Exercise 12: Securing RESTful Endpoints with Spring Security

1. **Add Spring Security Dependency:**

```
<!-- Add to pom.xml -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2. **JWT Authentication:**

Implement JWT authentication by creating JWT filter, provider, and security configuration.

```
// SecurityConfig.java
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
```

```
            .authorizeRequests()
            .antMatchers("/api/auth/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);
    }

    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter() {
        return new JwtAuthenticationFilter();
    }
}
```

3. **CORS Handling:**

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().and().csrf().disable();
    // Other configurations
}

@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("http://localhost:3000"));
    configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE"));
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

Exercise 13: Unit Testing REST Controllers

1. **JUnit Setup:**

```
<!-- Add to pom.xml -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

2. **MockMvc for Testing:**

```
@RunWith(SpringRunner.class)
@WebMvcTest(BookController.class)
public class BookControllerTest {
   @Autowired
   private MockMvc mockMvc;
   @MockBean
   private BookService bookService;
   @Test
   public void shouldReturnBook() throws Exception {
      BookDTO bookDTO = new BookDTO();
      bookDTO.setId(1L);
      bookDTO.setTitle("Spring Boot in Action");
      Mockito.when(bookService.getBook(1L)).thenReturn(bookDTO);
      mockMvc.perform(get("/api/books/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.title", is("Spring Boot in Action")));
   }
}
```

3. **Test Coverage:**

- Write tests for all edge cases, successful operations, and failure scenarios.

Exercise 14: Integration Testing for REST Services

1. **Spring Test Setup:**

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class BookServiceIntegrationTest {
   @Autowired
   private MockMvc mockMvc;
   @Test
   public void shouldReturnBook() throws Exception {
      mockMvc.perform(get("/api/books/1"))
            .andExpect(status().isOk());
   }
}
```

2. **MockMvc Integration:**

- Use @SpringBootTest and @AutoConfigureMockMvc to test REST endpoints.

3. **Database Integration:**

```
<!-- Add to pom.xml -->
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>test</scope>
</dependency>
```

Configure H2 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
```

Exercise 15: API Documentation with Swagger

1. **Add Swagger Dependency:**

```
<!-- Add to pom.xml -->
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.6.0</version>
</dependency>
```

2. **Document Endpoints:**

```
// BookController.java
@Tag(name = "books", description = "The Books API")
@RestController
@RequestMapping("/api/books")
public class BookController {
   @Operation(summary = "Get a book by its ID")
   @ApiResponses(value = {
         @ApiResponse(responseCode = "200", description = "Found the book",
               content = { @Content(mediaType = "application/json",
               schema = @Schema(implementation = BookDTO.class)) }),
         @ApiResponse(responseCode = "404", description = "Book not found", content = @Content)
   })
   @GetMapping("/{id}")
   public BookDTO getBook(@PathVariable Long id) {
      return bookService.getBook(id);
   }
}
```

3. **API Documentation:**

- Access the documentation at /swagger-ui.html or /v3/api-docs.