Spring Test - Unit Testing and Mocking

Unit testing is a fundamental practice in software development that involves testing individual, isolated components (or "units") of an application. In the context of a Spring Boot application, this means testing components like repositories, services, and controllers in isolation, without relying on the entire application context or external dependencies like a real database or web server.

To achieve this, we use a combination of Spring's testing framework and a mocking library, most commonly **Mockito**.

Part 1: Concepts

Unit Testing Concepts

- **Isolation**: The core principle of unit testing is to test a single "unit" of code in isolation. This means that when you test a service layer method, you shouldn't be making a real database call. The dependencies of the unit under test (e.g., a `Repository` in a `Service`) are replaced with "test doubles."
- **Test Doubles**: These are objects that stand in for real dependencies during a test. There are different types:
    - **Mocks**: Objects that simulate the behavior of a real object and verify that the tested code interacts with them as expected (e.g., verifying that a specific method was called).
    - **Stubs**: Objects that provide predefined answers to method calls. They're used to control the behavior of a dependency during a test.
    - **Fakes**: Objects that have a working implementation but are not suitable for production (e.g., an in-memory database).
- **Spring Test Slices**: Spring Boot provides specialized annotations to load only the necessary part of the application context for a specific layer. This makes tests faster and more focused.
    - `@DataJpaTest`: Loads the persistence layer, including JPA repositories and an in-memory database (like H2) for testing. This is ideal for testing the repository layer.
    - `@WebMvcTest`: Loads only the Spring MVC components, such as controllers, filters, and `MockMvc` for testing REST APIs. It doesn't load the service or repository layers.
- **Assertions**: Frameworks like JUnit and AssertJ provide assertion methods to verify the expected behavior of your code. For example, you can assert that a method returns a specific value, that a list has a certain size, or that an exception is thrown.

Mocking Concepts (with Mockito)

Mocking is the process of creating "mock" objects to simulate the behavior of real dependencies. This is crucial for unit testing the service and API layers, where you need to isolate the business logic from external systems.

- **`@Mock`**: This annotation is used to create a mock instance of a class or interface. It's used in conjunction with `@ExtendWith(MockitoExtension.class)` (for JUnit 5) or `@RunWith(MockitoJUnitRunner.class)` (for JUnit 4).
- **`@InjectMocks`**: This annotation creates an instance of the class you want to test and automatically injects the mock objects (created with `@Mock`) into it. This saves you from manually setting up the dependencies.
- **`@MockBean`**: This is a Spring-specific annotation that's used with Spring's test runner (`@SpringBootTest`, `@WebMvcTest`). It adds a mock object to the Spring application context, replacing any existing bean of the same type. This is useful when you're testing a component that relies on other beans managed by Spring, and you want to replace one of those beans with a mock.
- **Stubbing**: This is the process of defining the behavior of a mock object. The most common way to do this is with `Mockito.when(...)`.
  - `when(mockObject.methodCall()).thenReturn(value)`: This tells the mock to return a specific value when a certain method is called.
  - `when(mockObject.methodCall()).thenThrow(exception)`: This tells the mock to throw an exception when a method is called.
- **Verification**: This is the process of checking whether a method on a mock object was called, and how many times. The primary method is `Mockito.verify(...)`.
  - `verify(mockObject).methodCall()`: Verifies that the method was called exactly once.
  - `verify(mockObject, times(2)).methodCall()`: Verifies that the method was called twice.
  - `verify(mockObject, never()).methodCall()`: Verifies that the method was never called.

Part 2: Code Implementation

Let's assume we have a simple application with a `User` entity, a `UserRepository`, a `UserService`, and a `UserController`.

The Components to be Tested:

Java

```
// User.java
import javax.persistence.Entity;
```

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

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    // Getters, setters, constructors...
}
```
Java

```java
// UserRepository.java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User,
Long> {
    Optional<User> findByEmail(String email);
}
```
Java

```java
// UserService.java
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class UserService {

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
    public List<User> findAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> findUserById(Long id) {
        return userRepository.findById(id);
    }

    public User saveUser(User user) {
        // Simple business logic: don't save if email already
exists
        if
(userRepository.findByEmail(user.getEmail()).isPresent()) {
            throw new IllegalArgumentException("User with
this email already exists");
        }
        return userRepository.save(user);
    }
}
```
Java

```java
// UserController.java
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/users")
public class UserController {

    private final UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public List<User> getAllUsers() {
        return userService.findAllUsers();
    }
    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable
Long id) {
        Optional<User> user = userService.findUserById(id);
        return user.map(ResponseEntity::ok)
                   .orElseGet(() ->
ResponseEntity.notFound().build());
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User
user) {
        try {
            User savedUser = userService.saveUser(user);
            return ResponseEntity.ok(savedUser);
        } catch (IllegalArgumentException e) {
            return ResponseEntity.badRequest().build();
        }
    }
}
```

Unit Testing the Repository Layer (`@DataJpaTest`)

The repository layer is a good candidate for "slice" testing with a real in-memory database. We don't need to mock anything here because we want to test the actual JPA repository behavior.

Java

```
import org.junit.jupiter.api.Test;
import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTe
st;
import
org.springframework.boot.test.autoconfigure.orm.jpa.TestEntit
yManager;
import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository userRepository;

    @Test
    public void whenFindById_thenReturnUser() {
        // Given
        User user = new User();
        user.setName("John Doe");
        user.setEmail("john.doe@example.com");
        entityManager.persist(user);
        entityManager.flush();

        // When
        User found =
userRepository.findById(user.getId()).orElse(null);

        // Then
        assertThat(found).isNotNull();

assertThat(found.getName()).isEqualTo(user.getName());
    }

    @Test
    public void whenFindByEmail_thenReturnUser() {
        // Given
        User user = new User();
        user.setName("Jane Doe");
        user.setEmail("jane.doe@example.com");
        entityManager.persist(user);
        entityManager.flush();

        // When
        User found =
userRepository.findByEmail("jane.doe@example.com").orElse(nul
l);

        // Then
        assertThat(found).isNotNull();

assertThat(found.getEmail()).isEqualTo(user.getEmail());
    }

    @Test
    public void whenInvalidId_thenReturnNull() {
        // When
        User found =
userRepository.findById(100L).orElse(null);

        // Then
        assertThat(found).isNull();
    }
}
```

- `@DataJpaTest`: This annotation configures an in-memory database (like H2), sets up JPA, and scans for `@Repository` beans.
- `TestEntityManager`: This is provided by `@DataJpaTest` and is used to set up the test data directly in the database.
- The tests focus on verifying that the JPA repository methods (`findById`, `findByEmail`) work as expected with the database.

Unit Testing the Service Layer (with Mocking)

The service layer contains the business logic. We should unit test it in isolation, meaning we should mock its dependencies—in this case, the `UserRepository`.

Java

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
```

```
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    private User user1;
    private User user2;

    @BeforeEach
    public void setUp() {
        user1 = new User();
        user1.setId(1L);
        user1.setName("Alice");
        user1.setEmail("alice@example.com");

        user2 = new User();
        user2.setId(2L);
        user2.setName("Bob");
        user2.setEmail("bob@example.com");
    }

    @Test
    public void whenFindAllUsers_thenReturnUserList() {
        // Given
        List<User> userList = Arrays.asList(user1, user2);
        when(userRepository.findAll()).thenReturn(userList);

        // When
        List<User> foundUsers = userService.findAllUsers();

        // Then
        assertThat(foundUsers).hasSize(2);

assertThat(foundUsers.get(0).getName()).isEqualTo("Alice");
    }

    @Test
    public void whenFindUserById_thenReturnUser() {
        // Given

when(userRepository.findById(1L)).thenReturn(Optional.of(user
1));
```

```
        // When
        Optional<User> foundUser =
userService.findUserById(1L);

        // Then
        assertThat(foundUser).isPresent();

assertThat(foundUser.get().getName()).isEqualTo("Alice");
    }

    @Test
    public void whenSaveUser_thenSaveAndReturnUser() {
        // Given

when(userRepository.findByEmail(anyString())).thenReturn(Opti
onal.empty());

when(userRepository.save(any(User.class))).thenReturn(user1);

        // When
        User savedUser = userService.saveUser(user1);

        // Then
        assertThat(savedUser).isNotNull();
        assertThat(savedUser.getName()).isEqualTo("Alice");
        // Verify that the repository's save method was
called exactly once
        verify(userRepository, times(1)).save(user1);
    }

    @Test
    public void
whenSaveExistingUserEmail_thenThrowException() {
        // Given

when(userRepository.findByEmail("alice@example.com")).thenRet
urn(Optional.of(user1));

        // When & Then
        assertThrows(IllegalArgumentException.class, () ->
userService.saveUser(user1));
        // Verify that the save method was never called
        verify(userRepository,
never()).save(any(User.class));
```

```
    }
}
```

- `@ExtendWith(MockitoExtension.class)`: Integrates Mockito with JUnit 5.
- `@Mock UserRepository userRepository`: Creates a mock instance of the `UserRepository`.
- `@InjectMocks UserService userService`: Creates an instance of `UserService` and injects the `userRepository` mock into it.
- `when(...)`: This is where we stub the behavior of the mock repository. We define what the mock should return when a specific method is called.
- `verify(...)`: This is where we verify that the methods on the mock were called as expected. For example, in the `saveUser` test, we ensure that `userRepository.save()` was called exactly once.

Unit Testing the API Layer (`@WebMvcTest`)

For the controller layer, we want to test that the HTTP endpoints are correctly handled, without booting up the entire application or connecting to a real database. We use `@WebMvcTest` and mock the service layer.

Java

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.Test;
import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMv
cTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Arrays;
import java.util.Optional;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBu
ilders.get;
```

```
import static
org.springframework.test.web.servlet.request.MockMvcRequestBu
ilders.post;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatc
hers.jsonPath;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatc
hers.status;

@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private UserService userService;

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    public void whenGetAllUsers_thenReturnUserList() throws
Exception {
        // Given
        User user1 = new User();
        user1.setId(1L);
        user1.setName("Alice");


when(userService.findAllUsers()).thenReturn(Arrays.asList(use
r1));

        // When & Then
        mockMvc.perform(get("/api/users")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$[0].name").value("Alice
"));
    }

    @Test
    public void whenGetUserById_thenReturnUser() throws
Exception {
        // Given
        User user1 = new User();
        user1.setId(1L);
        user1.setName("Alice");


when(userService.findUserById(1L)).thenReturn(Optional.of(use
r1));

        // When & Then
        mockMvc.perform(get("/api/users/1")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value("Alice"))
;
    }

    @Test
    public void whenGetUserByInvalidId_thenReturnNotFound()
throws Exception {
        // Given

when(userService.findUserById(100L)).thenReturn(Optional.empt
y());

        // When & Then
        mockMvc.perform(get("/api/users/100")
                .contentType(MediaType.APPLICATION_JSON))
                .andExpect(status().isNotFound());
    }

    @Test
    public void whenCreateUser_thenReturnCreatedUser() throws
Exception {
        // Given
        User user = new User();
        user.setName("Charlie");
        user.setEmail("charlie@example.com");

        User savedUser = new User();
        savedUser.setId(1L);
        savedUser.setName("Charlie");
        savedUser.setEmail("charlie@example.com");
```

```
when(userService.saveUser(any(User.class))).thenReturn(savedU
ser);

        // When & Then
        mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(user
)))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.name").value("Charlie"
));
    }

    @Test
    public void
whenCreateUserWithExistingEmail_thenReturnBadRequest() throws
Exception {
        // Given
        User user = new User();
        user.setName("Charlie");
        user.setEmail("charlie@example.com");


when(userService.saveUser(any(User.class))).thenThrow(new
IllegalArgumentException());

        // When & Then
        mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(user
)))
                .andExpect(status().isBadRequest());
    }
}
```

- `@WebMvcTest(UserController.class)`: This annotation is a "slice" test for the web layer. It only loads the `UserController` and its dependencies (in this case, the `UserService`).

- `MockMvc`: This is the main class for testing Spring MVC controllers without a real HTTP server. It allows you to perform requests and assert on the response.

- `@MockBean private UserService userService`: This is a key part of the test. It adds a mock of `UserService`to the application context. When the `UserController` is instantiated, it will be injected with this mock `UserService`.

- The tests use `mockMvc.perform()` to simulate HTTP requests and `andExpect()` to verify the HTTP status, JSON content, and other aspects of the response.

Introduction to OpenAPI and the OpenAPI Specification (OAS)

OpenAPI is a powerful, vendor-neutral specification for describing RESTful APIs. It provides a standardized, language-agnostic interface that allows both humans and computers to understand an API's capabilities without needing to access its source code, documentation, or network traffic. The specification itself is called the OpenAPI Specification (OAS).

The OAS is a formal, machine-readable document (typically in YAML or JSON format) that acts as a contract for your API. It defines everything about the API, including its endpoints, operations, parameters, request and response bodies, data models, and security requirements. This single source of truth can be used throughout the entire API lifecycle, from design and development to testing, documentation, and consumption.

API Documentation

API documentation is a set of instructions and reference materials that developers use to understand and interact with an API. Well-structured API documentation is crucial for developer experience, as it allows users to quickly and effectively integrate an API into their applications.

The OpenAPI Specification is a cornerstone of modern API documentation. By defining an API using the OAS, you can leverage a wide array of tools to automatically generate interactive and user-friendly documentation. Tools like Swagger UI and ReDoc can take an OAS file and transform it into a visually appealing, browsable web page that includes all the necessary information for developers. This includes:

- A list of all available endpoints and their corresponding HTTP methods.
- Detailed descriptions of each operation.
- Examples of request and response bodies.
- Information on required parameters and their data types.
- Explanations of security and authentication methods.

API Paths and Operations

In OpenAPI, the structure of an API is defined through a combination of paths and operations.

- **Paths**: A path represents a specific endpoint or resource in your API. It's a relative URL from the server, such as `/users`, `/products/{id}`, or `/reports/summary/`. Paths can include placeholders for path parameters, like `{id}`, which are then defined in the operation.
- **Operations**: An operation corresponds to an HTTP method (GET, POST, PUT, DELETE, etc.) that can be performed on a specific path. Each operation is defined within a path and describes the action that the method performs. For example, for the `/users` path, you might have a `GET` operation to retrieve a list of users and a `POST` operation to create a

new user. Each operation object contains details like a summary, a description, parameters, and possible responses.

Implementation Example (YAML):

```
paths:
  /pets:
    get:
      summary: List all pets
      description: Retrieve a list of all pets in the store.
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time
(max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of pets
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
```

In this example, the path is `/pets`. The `get` operation defines what happens when a client makes a GET request to that path. It includes an optional query parameter (`limit`) and a definition for the successful response (status code `200`).

Request and Response Objects

These objects are used to define the data sent to and received from an API operation.

- **Request Object**: The `requestBody` object describes the data required to perform an operation that has a body (e.g., `POST`, `PUT`, or `PATCH`). It specifies the content type (e.g., `application/json`), the schema of the data, and whether the body is required.
- **Response Objects**: The `responses` object defines the possible outcomes of an API operation, mapped to HTTP status codes (e.g., `200` for success, `404` for not found, `500` for server error). Each response object describes the status code, a human-readable description, and the schema of the data returned in the response body. This is crucial for both documentation and for client-side code generation.

Implementation Example (YAML):

YAML

```
paths:
  /pets:
    post:
      summary: Create a pet
      requestBody:
        description: Pet to add to the store
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
      responses:
        '201':
          description: Pet created successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid input
```

Here, the `post` operation on the `/pets` path has a `requestBody` that requires a `Pet` object. It also defines a successful `201` response that returns the created `Pet` object and a `400` response for bad requests.

Data Models and Schemas

Data models and schemas are the blueprints for the data structures used in your API. The OpenAPI Specification uses the `Schema Object` to define data types and structures. Schemas are typically defined in the `components/schemas`section of the OAS file, which allows them to be reused across different paths, operations, and request/response objects.

A schema can describe simple types (like strings, integers, and booleans) or complex objects and arrays. You can also specify validation rules, such as `required` properties, `minimum`/ `maximum` values, and `enum` lists of acceptable values.

Implementation Example (YAML):

YAML

```
components:
  schemas:
    Pet:
      type: object
      required:
        - id
```

```
          - name
        properties:
          id:
            type: integer
            format: int64
            readOnly: true
          name:
            type: string
            example: 'doggie'
          tag:
            type: string
            description: A tag for the pet
```

This example defines a reusable `Pet` schema. It specifies that a `Pet` is an object with required `id` and `name` properties. The `id` is an integer and is marked as `readOnly`, meaning it will be returned in responses but should not be sent in a request body.

Security Definitions

Security is a critical part of any API. The OpenAPI Specification provides a way to describe security schemes and apply them to the entire API or to individual operations. Security definitions are handled in two parts:

1. **Security Scheme Object**: Defined under `components/securitySchemes`, this object describes the security mechanism itself. Supported types include:
    - `apiKey`: For API keys passed in headers, query parameters, or cookies.
    - `http`: For standard HTTP authentication schemes like Basic, Bearer, etc.
    - `oauth2`: For OAuth 2.0 flows, specifying authorization and token URLs, and scopes.
    - `openIdConnect`: For OpenID Connect discovery.
    - `mutualTLS`: For mutual TLS authentication.
2. **Security Requirement Object**: This object, defined at the global level or within a specific operation, references the security schemes and specifies which ones are required. For schemes like OAuth 2.0, it also indicates the necessary scopes.

Implementation Example (YAML):

YAML

```
openapi: 3.0.0
# ... other parts of the spec ...

security:
  - petstore_auth:
      - write:pets
      - read:pets

components:
  securitySchemes:
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: http://example.org/api/oauth/
dialog
          scopes:
            write:pets: modify pets in your account
            read:pets: read your pets
```

In this example, the `petstore_auth` security scheme is defined as an OAuth 2.0 implicit flow. The global `security`field then specifies that all operations require the `write:pets` and `read:pets` scopes. You can also override this at the operation level to require different or no security for specific endpoints.

. Paths and Operations

The `paths` object is the core of your API's structure in an OpenAPI document. It's a map where keys are URL paths and values are `Path Item Objects`. Each `Path Item Object` can contain a description and a list of operations.

- **Path Templating**: You can define dynamic segments in a path using curly braces, like `/users/{userId}/posts/{postId}`. These are called path parameters. Each path parameter must be explicitly defined in the `parameters` array of the corresponding path item or operation.
- **The `operationId`**: This is a crucial but often overlooked field. It provides a unique, case-sensitive string identifier for an operation. While optional, it's highly recommended for tools that generate client SDKs or server stubs, as it's often used to name the corresponding method. For example, `getUsers` or `createUser`.
- **The `tags` field**: This field is an array of strings that you can use to group related operations. Most documentation generators, like Swagger UI, use these tags to create logical groupings in the user interface, making large APIs easier to navigate.
- **Overriding Servers**: You can define a global `servers` array for your API's base URLs. However, you can also override this at the path or even the operation level. This is useful for APIs that have different base URLs for certain endpoints, for example, a media-specific CDN or a staging server.

2. Request and Response Objects

These objects go beyond simple data schemas to provide a complete picture of an API's data interaction.

- **`requestBody`**: The `requestBody` object for operations like `POST`, `PUT`, and `PATCH` is where you define the payload.
    - **`content`**: This is a map of media types (e.g., `application/json`, `application/xml`) to a `MediaType Object`. The `MediaType Object` contains the schema of the request body. This allows an API to support multiple data formats for the same endpoint.
    - **Examples**: You can provide concrete examples of the request body. This is invaluable for documentation and for tools that generate mock servers. Examples can be defined inline or as a reference to a separate example object in the `components/examples` section for reusability. You can even provide multiple examples with different names and descriptions.
- **`responses`**: This object is a map of HTTP status codes to `Response Objects`.
    - **Status Codes**: You can define specific status codes (`200`, `201`, `404`) and use wildcards (`2XX`, `5XX`) to cover ranges. The `default` key is a catch-all for any status code not explicitly defined.
    - **Headers**: Each `Response Object` can contain a `headers` object, which defines any custom HTTP headers that the API will return, such as `X-RateLimit-Limit` or `ETag`.
    - **Links**: A powerful feature of OpenAPI, `Link Objects` describe the relationship between different operations. For example, a `POST /users` operation might return a `201` response with a `Link` that points to the newly created user's resource at `GET /users/{userId}`. This is a form of HATEOAS (Hypermedia as an Engine of Application State).

3. Data Models and Schemas

OpenAPI schemas are based on a subset of the JSON Schema specification, but they extend it with additional keywords.

- **Reusability with `components/schemas`**: Defining your data models in the `components/schemas` section is a best practice. This centralizes your data structures, prevents duplication, and makes your OAS document more readable and maintainable. You reference these schemas using a `$ref` field, like `$ref: '#/components/schemas/Pet'`.
- **Schema Composition and Inheritance**: OpenAPI supports advanced schema composition techniques.
    - `allOf`: This combines multiple schemas into a single one. It's often used to simulate inheritance, where a new schema inherits all properties from one or more parent schemas.

    - `oneOf`: This specifies that the data must be a valid instance of **exactly one** of the schemas in the list. This is useful for polymorphic data where a field might be one of several different object types.
    - `anyOf`: Similar to `oneOf`, but the data can be a valid instance of **one or more** of the schemas.
    - `discriminator`: When using `oneOf` or `anyOf`, the `discriminator` field can be added to a schema to specify a property that will be used to determine which of the component schemas applies.
- **Validation Rules**: Schemas are not just for documentation; they provide a rich set of validation rules that can be enforced by tooling. These include:
    - `type`, `format`
    - `required` properties
    - String constraints: `minLength`, `maxLength`, `pattern` (regex)
    - Numeric constraints: `minimum`, `maximum`, `exclusiveMinimum`, `exclusiveMaximum`
    - Array constraints: `minItems`, `maxItems`, `uniqueItems`
    - Object constraints: `minProperties`, `maxProperties`

4. Security Definitions

OpenAPI security definitions are the formal way to describe how your API is protected.

- **`components/securitySchemes`**: This is where you declare the security mechanisms, giving each a unique name.
    - `type`: `apiKey` - `name` and `in` fields specify the key's name and its location (`query`, `header`, or `cookie`).
    - `type`: `http` - `scheme` field specifies the authentication scheme, like `Basic` or `Bearer`. For `Bearer`, the `bearerFormat` field can provide more information, like `JWT`.
    - `type`: `oauth2` - `flows` object defines the specific OAuth 2.0 flow (e.g., `authorizationCode`, `clientCredentials`). You must provide `authorizationUrl`, `tokenUrl`, and `scopes` for each flow.
- **`security` Field**: This field links the declared security schemes to the API's operations.
    - **Global Security**: Defined at the root of the document, it applies to all operations unless overridden. It's an array of objects, where each object is a map of a security scheme name to an array of required scopes.

- **Operation-Level Security**: This is defined within a specific operation object. It overrides the global security for that single operation. You can set it to an empty array (`[ ]`) to make an operation publicly accessible, even if a global security requirement exists. You can also specify multiple security schemes to indicate that an operation can be accessed with *either* one.