Java and Spring Boot: Core Concepts

1. Dependency Injection and Inversion of Control

1.1 Dependency Injection

Dependency injection provides objects that an object needs (its dependencies) instead of having it construct them itself. This technique is particularly useful for testing, as it allows dependencies to be mocked or stubbed out.

1.2 Constructor Dependency Injection

Dependencies can be injected via constructor using the <constructor-arg> subelement of <bean>. This allows injection of:

- Primitive and String-based values
- Dependent objects (contained objects)
- Collection values

2. Spring Bean Lifecycle

2.1 Container Startup

The Spring container starts and loads bean definitions from configuration files or annotated classes.

2.2 Bean Instantiation

Spring creates instances of beans using constructors or factory methods.

2.3 Bean Initialization

After dependency injection, custom initialization logic is executed (via @PostConstruct, init-method, or afterPropertiesSet()).

2.4 Bean Ready For Use

The bean is fully initialized and ready to be used within the application.

2.5 Bean Destruction

Before the container shuts down, Spring executes custom cleanup logic (via @PreDestroy, destroy-method, or destroy()).

2.6 Container Shutdown

The Spring container shuts down, releasing resources and cleaning up all beans.

3. Key Spring Annotations and Concepts

3.1 Core Annotations

- **@SpringBootApplication**: class marker, combines @Configuration, @EnableAutoConfiguration, and @ComponentScan

3.2 Specialized Stereotypes

- **@Service**: Implements business logic in service layer
- **@Repository**: Marks data access layer classes
- **@Controller**: Handles web requests in MVC pattern

3.3 Dependency Injection Annotations

- **@Autowired**:
  - Constructor Injection (preferred)
  - Field Injection
  - Setter Injection
- Optional Parameter Injection using required = false

3.4 Web-Related Annotations

- **@Bean**: Declares Spring-managed beans
- **@RestController**: Combines @Controller and @ResponseBody
- **@RequestMapping**: Maps HTTP requests to handlers

4. JPA (Java Persistence API)

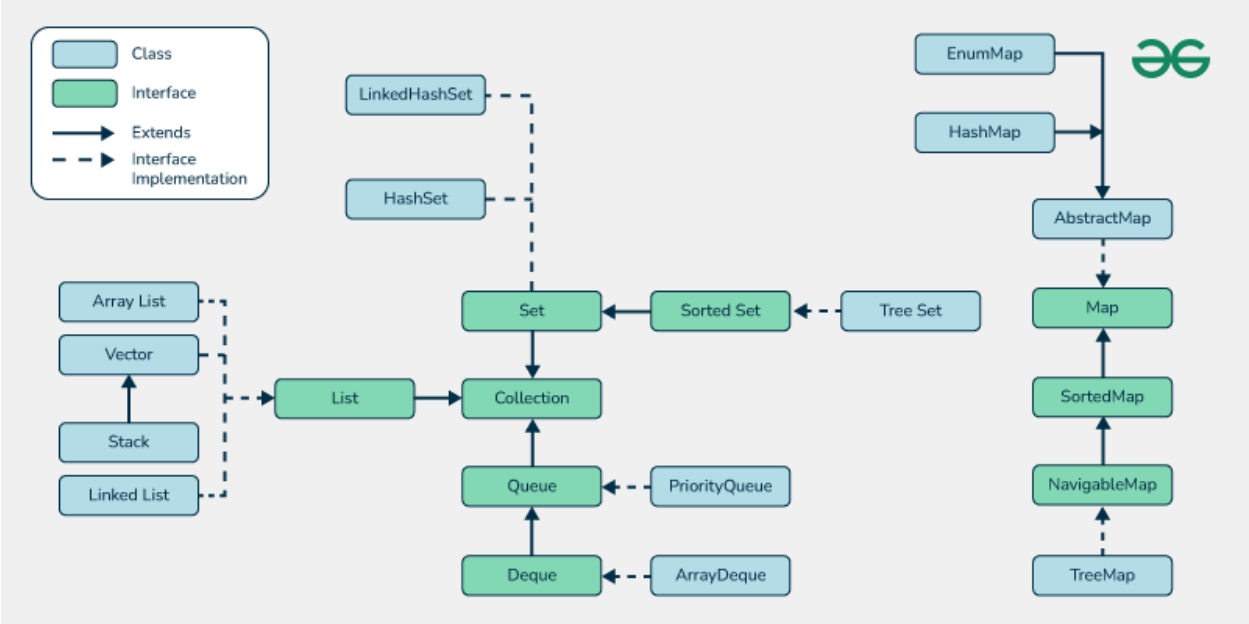
Spring Data JPA simplifies data access layer implementation by reducing boilerplate code. It provides:

- Easy repository implementation
- Built-in pagination
- Auditing capabilities

4.1 Key JPA Annotations

- **@Entity**: Marks JPA entities
- **@Table**: Specifies table details
- **@Id**: Marks primary keys
- **@GeneratedValue**: Configures key generation

5. Java Collections Framework



6. Maven: Build and Dependency Management

6.1 Key Features

- **Build Automation**: Automates the build process for Java-based projects.
- **Dependency Management**: Simplifies adding, updating, and managing project dependencies.
- **Project Configuration via pom.xml**: Centralized configuration using the `pom.xml` file.

6.2 Maven Workflow

1. **Project Definition**: Define the project's structure and metadata in `pom.xml`.

2. **Dependency Declaration**: Add required libraries and frameworks as dependencies in `pom.xml`.
3. **Command Execution**: Run commands (e.g., `mvn clean`, `mvn package`) to perform specific tasks.
4. **Dependency Resolution**: Automatically downloads and resolves dependencies from Maven repositories.
5. **Project Building**: Compiles, tests, and packages the project into a deployable format.

6.3 POM (Project Object Model)

Contains:

- Project information
- Dependencies
- Build configuration
- Repository locations

6.4 Default Lifecycle Phases

1. **validate**: Verifies if the project is correctly configured and all required resources are available.
2. **compile**: Compiles the source code of the project.
3. **test**: Runs unit tests to ensure the code works as expected.
4. **package**: Packages the compiled code into a distributable format (e.g., JAR, WAR).
5. **install**: Installs the packaged artifact into the local Maven repository for use in other projects.
6. **deploy**: Uploads the packaged artifact to a remote repository for sharing with other developers or systems.

7. Object-Oriented Programming Concepts

7.1 Abstract Class vs Interface Comparison

Abstract Class

- Can have both abstract and concrete methods
- Supports instance variables and constructors
- Single inheritance only
- Various access modifiers supported

Interface

- Contains primarily abstract methods
- Only constants (static final) allowed
- Multiple inheritance supported
- Methods public by default

7.2 Core OOP Principles

1. **Encapsulation**: Data and method bundling
2. **Inheritance**: Property and method inheritance
3. **Polymorphism**: Multiple forms of methods/objects
4. **Abstraction**: Implementation hiding

8. Security Implementation

8.1 Password Hashing

Includes implementation examples for:

- **SHA Hashing**:

  SHA (Secure Hash Algorithm) is a cryptographic hash function used to convert passwords into fixed-size hash values.

```java
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] hashedPassword = md.digest(password.getBytes(StandardCharsets.UTF_8));
```

- **BCrypt Password Encoding**:

  BCrypt is a stronger, adaptive hashing algorithm that includes a salt for additional security. It is widely used for password storage.

```java
String hashedPassword = BCrypt.hashpw(password, BCrypt.gensalt());
```

- **Password Validation**:
- Password validation involves comparing the user's input with the stored hashed password.

```java
boolean matches = BCrypt.checkpw(password, storedHashedPassword);
```

8.2 Spring Security Overview

- **Authentication**:
  Verifies the identity of a user, typically through username and password, ensuring the user is who they claim to be.

- **Authorization**:
  Determines if an authenticated user has permission to access specific resources or perform certain actions.

- **Session Management**:
  Manages user sessions, handling things like session creation, timeouts, and preventing session fixation attacks.

- **CSRF Protection**:
  Protects against Cross-Site Request Forgery (CSRF) attacks by ensuring that requests to sensitive operations are sent by the authenticated user, using tokens to validate each request.

8.3 Cross-Site Scripting (XSS) Protection

- **Stored XSS**:
  Malicious scripts are permanently stored on the server (e.g., in a database) and are executed when other users view the affected page.

- **Reflected XSS**:
  Malicious scripts are embedded in a URL and executed immediately after being reflected back by the server in an error message, search

result, or any other response.

- **DOM-based XSS**:
  Malicious scripts are executed when the browser processes data (e.g., from the URL or user input) on the client-side, without involving the server directly.

Protection Methods:

- **Input sanitization**: Ensuring that input does not contain executable scripts.
- **Escaping**: Converting potentially dangerous characters into safe equivalents (e.g., `<` to `<`).
- **Content Security Policy (CSP)**: Enforcing rules about where content can be loaded from and preventing inline scripts.

9. HTTP Session Management

9.1 Cookie-Based Sessions

Process Flow:

- **Session Creation**:
  When a user first accesses the application, a unique session ID is generated on the server and associated with the user's session.

- **Session ID Transmission**:
  The session ID is sent from the server to the browser via a

Set-Cookie header. The browser stores this session ID as a cookie.

- **Browser Storage**:
  The browser stores the session ID in a cookie, which is sent with each subsequent request to the server.

- **Session Validation**:
  On each request, the server checks the session ID sent in the cookie. If valid, the session is authenticated, and the user is identified.

- **Session Termination**:
  When the user logs out or the session expires, the session is invalidated, and the session ID is removed from the cookie.

```
JSESSIONID=3D2F1D9A30E8A2B1C4F4C5D6D1E5F6A9; Path=/; HttpOnly; Secure; SameSite=Strict
```

Set-Cookie: JSESSIONID=9F1C2A2C0D51A348F85B229A6B8E5576; Path=/; HttpOnly; Secure; SameSite=Strict

10. Testing Frameworks

Key Annotations:

- **@Test**:
  Marks a method as a test method to be executed by the JUnit framework.
- **@BeforeEach**:
  Indicates that the annotated method should be run **before each test** method, used for setup tasks.

- **@AfterEach**:
  Indicates that the annotated method should be run **after each test** method, used for cleanup tasks.
- **@BeforeAll**:
  Indicates that the annotated method should be run **once before all test methods** in the class, often used for expensive setup (must be static in JUnit 5).
- **@AfterAll**:
  Indicates that the annotated method should be run **once after all test methods** in the class, often used for cleanup (must be static in JUnit 5).
- **@Disabled**:
  Marks a test method or class to be ignored (disabled) during test execution.

Mockito

Features:

- **Mock Object Creation**:
  Allows creating mock objects of classes or interfaces to simulate real objects in unit tests.
- **Behavior Definition**:
  Defines how the mock objects should behave when certain methods are called using `when()` and `thenReturn()`.
- **Interaction Verification**:
  Verifies if a specific method was called on a mock object using `verify()` to ensure correct interactions.

- **Exception Testing**:
  Allows testing how methods handle exceptions by using `when()` to throw exceptions from mock objects and verifying the expected behavior.

11. Git Version Control

11.1 Core Concepts

Repository:
A storage space for your project, containing all files, commit history, and branches. It can be local (on your computer) or remote (on a server like GitHub).

Commit:
A snapshot of changes made to the repository at a specific point in time, recorded with a unique ID.

Branch:
A parallel version of the repository, used to work on different features or fixes without affecting the main codebase (usually `main` or `master`).

Merge:
The process of integrating changes from one branch into another, usually from a feature branch into the main branch.

Clone/Push/Pull:

- **Clone**: Creates a local copy of a remote repository.

- **Push**: Uploads local changes to the remote repository.
- **Pull**: Fetches and merges changes from the remote repository to the local repository.

Fetch:

Downloads new commits and data from the remote repository but does not automatically merge them into your current branch.

Status:

Shows the current state of the working directory, including changes that are staged for commit, changes not staged, and untracked files.

11.2 Common Git Commands

- **git init**:
  Initializes a new Git repository in the current directory, creating a `.git` directory.

- **git add**:
  Stages changes (new, modified, or deleted files) to be committed in the next commit. Example: `git add .` (adds all changes).

- **git commit**:
  Records the staged changes as a snapshot in the repository with a message describing the changes. Example: `git commit -m "Commit message"`.

- **git push/pull**:

    - **git push**: Uploads local commits to a remote repository.
    - **git pull**: Fetches and merges changes from the remote repository into the local repository.
- **git branch**:
  Lists, creates, or deletes branches. Example: `git branch <branch-name>` creates a new branch.

- **git checkout**:
  Switches between branches or restores files. Example: `git checkout <branch-name>` switches to a branch, or `git checkout -- <file>` restores a file from the last commit.

- **git merge**:
  Merges change from one branch into the current branch. Example: `git merge <branch-name>` merges the specified branch into the current branch.