Chapter 1: JDBC and Maven

1. Explain the use of Maven, and its life cycle.

Answer:

Maven is a build automation tool used primarily for Java projects. It simplifies the build process, dependency management, and project structure.

Maven Life Cycle: Maven has three main life cycles:

1. **Default Life Cycle**: This handles the project deployment and is the most commonly used.
    - **Phases**: validate, compile, test, package, verify, install, deploy.
2. **Clean Life Cycle**: It handles the removal of files from the previous build.
    - **Phases**: pre-clean, clean, post-clean.
3. **Site Life Cycle**: It generates project documentation.
    - **Phases**: pre-site, site, post-site.

Each phase represents a step in the build process. Phases are executed in a specific order, and each phase has one or more goals associated with it.

2. What is JDBC? Write its different interfaces and API with their uses.

Answer:

JDBC (Java Database Connectivity) is an API for connecting and executing queries in a database.

JDBC Interfaces and their Uses:

- **Connection**: Manages the connection to the database.
- **Statement**: Executes SQL queries.
- **PreparedStatement**: Extends Statement to execute precompiled SQL statements with or without parameters.
- **ResultSet**: Represents the result set of a query.
- **CallableStatement**: Used to execute stored procedures in the database.
- **Driver**: Handles the connection between JDBC and the database.
- **DataSource**: Provides a more flexible and efficient way to get connections.

3. What is DriverManager? Explain different types of DriverManagers available.

Answer:

DriverManager is a class in JDBC that manages a list of database drivers. It establishes a connection to the database by selecting an appropriate driver.

Types of JDBC Drivers:

Type	Description
Type 1: JDBC-ODBC Bridge Driver	Converts JDBC calls into ODBC calls.
Type 2: Native-API Driver	Uses DBMS-specific native APIs to communicate with the database.
Type 3: Network Protocol Driver	Uses a middleware server to handle communication with the database.
Type 4: Thin Driver	Directly converts JDBC calls into database-specific protocol.

4. Draw diagram of 2-tier and 3-tier architecture of JDBC.

Answer:

- **2-tier architecture**:
    - Client <-> Database (Direct connection)

Diagram:

arduino

CODE

Client <---> Database

- **3-tier architecture**:
    - Client <-> Middle Tier (Application Server) <-> Database

Diagram:

arduino

CODE

Client <---> Application Server <---> Database

5. Explain the different types of JDBC drivers.

Answer:

Type of Driver	Description	Pros	Cons
Type 1: JDBC-ODBC Bridge	Converts JDBC calls to ODBC calls.	Easy to use, works with any DBMS.	Slow, relies on ODBC drivers.
Type 2: Native-API	Uses DBMS-specific API to connect.	Fast, native to DBMS.	Platform-dependent, requires native libraries.
Type 3: Network Protocol	Uses middleware server for communication.	Database-independent, can work with multiple DBs.	Requires an extra layer (middleware).

Type of Driver	Description	Pros	Cons
Type 4: Thin Driver	Directly communicates with the DBMS using its native protocol.	Very fast, platform-independent.	DBMS-specific, only works with certain databases.

6. Write the steps to connect MySQL database with your code.

Answer:

1. **Load the MySQL JDBC Driver**:

java

CODE

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

2. **Create Connection**:

java

CODE

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/mydb", "username", "password");
```

3. **Create Statement Object**:

java

CODE

```
Statement stmt = con.createStatement();
```

4. **Execute Query**:

java

CODE

```
ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
```

5. **Process Results**:

java

CODE

```
while (rs.next()) {
  System.out.println(rs.getString(1));
}
```

6. **Close Connection**:

java

CODE

con.close();

7. Explain the following terms:

- **a) ResultSet**: It represents the result set of a query. It is used to iterate over the data retrieved from a database.
- **b) Query**: A query is a SQL statement used to retrieve data from a database.
- **c) PreparedStatement**: An interface that represents a precompiled SQL statement which can accept parameters.
- **d) Connection**: An interface that represents the communication link between the Java application and the database.
- **e) Statement**: An interface that allows executing SQL queries against the database.
- **f) POM (Project Object Model)**: It is an XML file in Maven that contains information about the project such as dependencies, build configurations, and plugin settings.

Chapter 2: Servlets and HTTP

8. Write the uses of web.xml file.

Answer:

The web.xml file, also known as the **deployment descriptor**, is used for configuring the web application in Java. It contains mappings, initialization parameters, security configurations, and other settings.

Uses:

- Configures Servlets, Filters, and Listeners.
- Maps URLs to servlets.
- Configures initialization parameters for servlets.
- Defines security settings (e.g., user roles, authentication).
- Configures error pages.
- Maps the dispatcher types for filters.

9. List the various HTTP status codes and their messages.

Answer:

Status Code	Message	Description
200	OK	The request was successful.

Status Code	Message	Description
301	Moved Permanently	The resource has been permanently moved to a new URL.
400	Bad Request	The server cannot process the request due to client error.
401	Unauthorized	Authentication is required.
403	Forbidden	The server understands the request but refuses to authorize it.
404	Not Found	The requested resource could not be found.
500	Internal Server Error	The server encountered an error and could not complete the request.

10. What is Servlet? List its different interfaces and classes with their application.

Answer:

A **Servlet** is a Java class that runs on a server and processes requests and generates responses. It is used to extend the functionality of web servers.

Servlet Interfaces and Classes:

Interface/Class	Description
Servlet	Defines methods like init(), service(), destroy() for handling requests.
GenericServlet	An abstract class that simplifies the Servlet API by providing default implementations.
HttpServlet	A subclass of GenericServlet that provides methods for handling HTTP requests (doGet(), doPost()).
ServletRequest	Provides methods to handle incoming requests.
ServletResponse	Provides methods to send responses to the client.
ServletContext	Provides information about the environment of the servlet.
ServletConfig	Provides configuration for the servlet.

11. Differentiate between the doGet and doPost method of HTTP Servlet.

Feature	doGet	doPost
Purpose	Handles HTTP GET requests (retrieve data).	Handles HTTP POST requests (submit data).
Data Handling	Sends data in the URL (query string).	Sends data in the request body.

Feature	doGet	doPost
Visibility	Data is visible in the URL.	Data is not visible in the URL.
Caching	Caching is possible.	Caching is not recommended.
Security	Less secure (data visible in URL).	More secure (data not visible).
Use Case	When data retrieval is required.	When data submission or modification is required.

12. Explain the Servlet Life Cycle in details.

Answer:

The Servlet Life Cycle includes the following stages:

1. **Loading the Servlet**: The servlet is loaded into memory when the first request is made.
2. **Initialization**: The init() method is called once to initialize the servlet.
3. **Request Handling**: The service() method is called for each client request. It processes the request and generates a response.
4. **Destruction**: When the servlet is no longer required, the destroy() method is called to clean up resources

13. Explain the methods such as init(), service(), and destroy().

Answer:

Method	Description
init()	This method is called once when the servlet is loaded into memory. It is used for initialization, such as setting up resources (e.g., database connections).
service()	This method is called for each client request. It processes the request and generates the appropriate response. It takes ServletRequest and ServletResponse objects as parameters.
destroy()	This method is called just before the servlet is destroyed (i.e., when the server shuts down or the servlet is unloaded). It is used to release resources (e.g., closing database connections).

14. What is the javax.servlet package? List javax.servlet and explain its classes and interfaces of the package.

Answer:

The javax.servlet package provides classes and interfaces to create servlets and handle HTTP requests and responses.

Key Classes and Interfaces in javax.servlet package:

Class/Interface	Description
Servlet	The base interface for creating servlets. Defines methods like init(), service(), and destroy().
GenericServlet	A convenience class that implements Servlet interface and provides default implementations.
HttpServlet	A subclass of GenericServlet designed for HTTP-specific functionality (provides methods like doGet() and doPost()).
ServletRequest	Represents the client's request. Provides methods to get request parameters, headers, and other data.
ServletResponse	Represents the response to be sent to the client. Provides methods to send content, headers, etc.
ServletContext	Provides access to the servlet container's environment (e.g., web application resources).
ServletConfig	Provides configuration data specific to a servlet (e.g., initialization parameters).
Filter	An interface for filtering incoming requests and outgoing responses.
FilterChain	Used to pass request and response to the next filter or servlet in the chain.
Listener	Interface for objects that respond to lifecycle events such as contextInitialized() or sessionCreated().

Chapter 3: JSP (JavaServer Pages)

15. What is JSP and why do we need it?

Answer:

JSP (JavaServer Pages) is a server-side technology that allows developers to create dynamic, web-based content. It is used to generate HTML, XML, or other types of documents in response to a client request.

Why JSP?

1. Allows embedding Java code directly within HTML pages.
2. Simplifies the creation of dynamic web pages (contrast with servlets, which require more code).
3. JSPs are compiled into servlets, allowing for performance optimization.
4. Supports separation of concerns (designers and developers can work independently).
5. Provides tag libraries (JSTL) for reusable components.

16. What are JSP life cycle phases?

Answer:

JSP Life Cycle involves the following phases:

1. **Translation Phase**: The JSP file is converted into a servlet by the JSP engine.
2. **Compilation Phase**: The generated servlet is compiled into a bytecode.
3. **Initialization Phase**: The init() method of the generated servlet is called.
4. **Execution Phase**: For each request, the service() method is called to process the request and generate a response.
5. **Destroy Phase**: When the servlet is no longer needed, the destroy() method is invoked to clean up resources.

17. Method of JSP

Answer:

In JSP, there are several methods used during the life cycle:

1. **_jspService()**: This method is called for each HTTP request. It is where the actual response is generated from the JSP content. This method cannot be overwritten.
2. **_jspInit()**: It is used to initialize the JSP page; this method is automatically called when the JSP page is loaded for the first time.
3. **_jspDestroy()**: This method is called when the JSP page is being destroyed or unloaded. It is used to release any resources.

18. JSP Scripting and JSP Scriplet

Answer:

JSP scripting elements allow embedding Java code directly within the HTML page.

- **Scriptlet (<% ... %>)**: It allows embedding Java code between <% and %>. Example:

jsp

CODE

```
<%
  String name = "John";
  out.println("Hello, " + name);
%>
```

- **Expression (<%= ... %>)**: It evaluates the Java code and prints the result to the client. Example:

jsp

CODE

```
<%= 5 + 3 %> <!-- Prints 8 -->
```

- **Declaration (<%! ... %>)**: It is used to declare variables or methods within the JSP page. Example:

jsp

CODE

<%! int counter = 0; %>

19. CRUD Operation (Not program)

Answer:

CRUD stands for Create, Read, Update, and Delete. These are the basic operations for managing data in a database.

- **Create**: Insert new records into the database (e.g., INSERT SQL).
- **Read**: Retrieve existing records from the database (e.g., SELECT SQL).
- **Update**: Modify existing records in the database (e.g., UPDATE SQL).
- **Delete**: Remove records from the database (e.g., DELETE SQL).

Chapter 4: Hibernate and JPA

1. What is Hibernate, and why is it used?

Answer:

Hibernate is an open-source Object-Relational Mapping (ORM) framework that provides a framework to map Java objects to database tables. It simplifies database interactions, eliminating the need for manual SQL queries.

Why Hibernate?

1. Automates database interactions.
2. Provides caching, reducing database load.
3. Supports complex queries and relationships.
4. Simplifies transaction management.
5. Reduces development time by eliminating boilerplate code.

2. What are Hibernate's core interfaces?

Answer:

- **Session**: Used to interact with the database and perform CRUD operations.
- **SessionFactory**: A factory for creating Session objects.
- **Transaction**: Manages database transactions.
- **Query**: Represents a HQL (Hibernate Query Language) query.

- **Criteria**: Provides an API for building database queries in an object-oriented way.
- **Configuration**: Configures Hibernate settings such as database connection and mapping.

3. Explain the life cycle states of a Hibernate object.

Answer:

1. **Transient State**: The object is created but not yet associated with the session (not persisted in the database).
2. **Persistent State**: The object is associated with a session and mapped to a row in the database.
3. **Detached State**: The object is no longer associated with the session, but its state is still available (e.g., after closing the session).
4. **Removed State**: The object is marked for deletion from the database.

4. How do Hibernate mappings work?

Answer:

Hibernate mappings define how Java objects (entities) are mapped to database tables. Mappings are configured either through annotations or XML files.

- **Annotations**:
    - @Entity marks a class as a Hibernate entity.
    - @Id marks the primary key of the entity.
    - @ManyToOne, @OneToMany, etc., define relationships.
- **XML Mappings**: Use <hibernate-mapping> tags to map entities and relationships.

5. What is the difference between save(), persist(), update(), and merge() in Hibernate?

Answer:

Method	Description
save()	Persists a new object. The object is assigned a generated identifier and added to the session.
persist()	Similar to save(), but doesn't return the identifier and is more aligned with JPA specifications.
update()	Updates an existing object in the database. It throws an exception if the object does not exist in the database.
merge()	Merges the state of a detached object with the current session, preserving the original object if it's already managed.

6. Explain caching in Hibernate.

Answer:

Hibernate uses two levels of caching:

1. **First-Level Cache**:
    - A session-level cache.
    - Enabled by default.
    - The cache is cleared once the session is closed.
2. **Second-Level Cache**:
    - A session factory-level cache.
    - Can be configured to persist objects across sessions.
    - Typically used for frequently accessed data.

7. What is HQL (Hibernate Query Language) and how does it differ from SQL?

Answer:

HQL is an object-oriented query language that is similar to SQL but operates on the entity objects instead of database tables.

- **Difference from SQL**:
    1. HQL uses entity names, not table names.
    2. HQL uses class properties, not column names.
    3. HQL supports polymorphic queries (fetching subclasses).

8. Describe the @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany relationships in Hibernate.

Answer:

- **@OneToOne**: A single entity is associated with one other entity.
- **@OneToMany**: A single entity is associated with multiple entities.
- **@ManyToOne**: Many entities are associated with a single entity.
- **@ManyToMany**: Many entities are associated with many other entities.

9. How does Hibernate handle transactions?

Answer:

Hibernate uses **JTA (Java Transaction API)** or **JDBC transactions** to manage transactions:

1. Begin a transaction using session.beginTransaction().
2. Perform the database operations.
3. Commit the transaction using transaction.commit().
4. If an error occurs, use transaction.rollback() to undo changes.

Chapter 5: Spring Framework

1. What is the Spring Framework, and what are its key modules?

Answer:

Spring is an open-source framework that provides comprehensive infrastructure support for Java applications. It facilitates the development of Java-based enterprise applications by providing various modules.

Key Modules:

1. **Core Container**: Provides fundamental functionality like dependency injection.
2. **AOP (Aspect-Oriented Programming)**: Supports cross-cutting concerns (logging, transaction management).
3. **Data Access/Integration**: JDBC, ORM (Hibernate), JMS, transactions.
4. **Web**: Spring MVC, RESTful services.
5. **Security**: Handles authentication and authorization.
6. **Test**: Simplifies the testing of Spring components.

2. Explain Dependency Injection (DI) and Inversion of Control (IoC) in Spring.

Answer:

- **Inversion of Control (IoC)**: A design principle where the control of object creation and management is transferred to the Spring container rather than the application code.
- **Dependency Injection (DI)**: A specific form of IoC where dependencies (objects required by a class) are injected by the Spring container, typically through constructor, setter, or field injection.

3. What are the different types of dependency injection in Spring?

Answer:

1. **Constructor Injection**: Dependencies are passed through the constructor.
2. **Setter Injection**: Dependencies are provided through setter methods.
3. **Field Injection**: Dependencies are injected directly into fields (less common).

---

4. Explain the Spring Bean lifecycle.

Answer:

The **Spring Bean lifecycle** includes several stages:

1. **Bean Instantiation**: Spring creates an instance of the bean (either via constructor or factory method).
2. **Dependency Injection**: Spring injects any dependencies (other beans) into the bean.

3. **Bean Post Processors**: Before and after the bean's initialization method is called, Spring invokes BeanPostProcessor methods for additional custom processing.
4. **Initializing Bean**: If the bean implements InitializingBean, the afterPropertiesSet() method is called, or a custom init-method is invoked (configured in XML or annotations).
5. **Bean Ready for Use**: The bean is now ready to be used by the application.
6. **Destroying Bean**: If the bean implements DisposableBean, the destroy() method is called, or a custom destroy-method is invoked when the application context is closed.

5. What are the scopes of Spring beans?

Answer:

Spring provides several scopes for beans, which define the lifespan of the bean within the container:

1. **Singleton**: A single instance of the bean is created and shared across the entire application.
2. **Prototype**: A new instance of the bean is created every time it is requested.
3. **Request**: A new bean instance is created for each HTTP request (used in web applications).
4. **Session**: A new bean instance is created for each HTTP session.
5. **Application**: A bean is scoped to a web application context, shared across multiple HTTP sessions.
6. **WebSocket**: A new bean instance is created for each WebSocket session.

6. How does Spring handle cross-cutting concerns with AOP (Aspect-Oriented Programming)?

Answer:

AOP (Aspect-Oriented Programming) in Spring allows developers to separate concerns such as logging, security, or transaction management from the core business logic. Spring AOP works by applying **aspects** to **join points** (points in the execution flow, like method calls). This is achieved using **advice** and **pointcuts**:

- **Advice**: Code that runs at a specific join point. Types of advice:
    - @Before: Executes before the method runs.
    - @After: Executes after the method runs, regardless of the outcome.
    - @AfterReturning: Executes after the method runs, only if successful.
    - @AfterThrowing: Executes if the method throws an exception.
    - @Around: Wraps the method execution, allowing modification of input/output and control over whether the method is executed.
- **Pointcut**: Specifies where the advice should be applied (e.g., methods in a specific class).

Spring AOP is often used for managing **cross-cutting concerns** like logging, transaction management, or security.

7. What is the role of @Transactional in Spring?

Answer:

The @Transactional annotation in Spring is used to handle transaction management in a declarative manner. It indicates that a method or class should run within a transaction context. The transaction is automatically started before the method execution and committed (or rolled back) after the method completes.

Key Features:

- Ensures **atomicity**, **consistency**, **isolation**, and **durability (ACID)** properties of transactions.
- Supports rollback on exceptions (default is rollback on unchecked exceptions).
- Can be applied to both service layer methods and at class level to affect all methods.

Example:

java

CODE

```
@Transactional
public void transferMoney(Account from, Account to, double amount) {
  from.debit(amount);
  to.credit(amount);
}
```

8. Explain the differences between @Controller and @RestController.

Answer:

Feature	**@Controller**	**@RestController**
Purpose	Used for handling traditional web requests in MVC applications.	Used for creating RESTful web services (returning JSON or XML).
Response Type	Returns a view (typically JSP or Thymeleaf).	Automatically returns data (usually JSON or XML).
Usage	Typically used in applications with a front-end view layer.	Used for APIs, returning data rather than views.
View Resolution	Uses a view resolver to return a view (HTML page).	Does not use views. The data is returned as the response body.

@RestController is a specialized version of @Controller with @ResponseBody automatically applied to all methods.

9. What are Spring Profiles, and how are they used?

Answer:

Spring Profiles allow you to define multiple configurations for different environments (e.g., development, testing, production) within a Spring application. Profiles help to separate and load configurations based on the environment the application is running in.

Usage:

- You can define beans that are specific to certain profiles using @Profile.
- Profiles can be activated through configuration files (application.properties) or programmatically.

Example:

java

CODE

```
@Profile("dev")
@Bean
public DataSource dataSource() {
   return new H2DataSource();
}


@Profile("prod")
@Bean
public DataSource dataSource() {
   return new MySQLDataSource();
}
```

Activating a Profile:

properties

CODE

```
# In application.properties
spring.profiles.active=dev
```

10. How do you configure Spring with XML vs. annotations?

Answer:

Aspect	XML Configuration	Annotation-Based Configuration
Configuration File	applicationContext.xml or other XML files.	Use annotations such as @Component, @Bean, @Autowired.

Aspect	XML Configuration	Annotation-Based Configuration
Bean Definition	Beans are defined explicitly in XML files.	Beans are defined with @Component, @Service, @Repository.
Dependency Injection	Injected using <bean> and autowire attributes in XML.	Use @Autowired annotation for dependency injection.
Complexity	More verbose and XML-centric.	Easier and more concise with annotations.
Flexibility	More flexibility in configuration, as it's externalized.	Configuration is part of the code itself.
Support for Profiles	Profiles can be defined in XML using <beans profile="...">.	Use @Profile to define beans for specific environments.

Chapter 6: Spring Boot

1. What is Spring Boot, and how does it differ from Spring?

Answer:

Spring Boot is a framework built on top of the Spring Framework that simplifies the setup and configuration of Spring applications. It eliminates the need for extensive configuration files and provides an embedded web server to run applications directly.

Differences:

- **Configuration**: Spring Boot reduces boilerplate code and configuration (no need for XML or heavy configuration files).
- **Standalone Applications**: Spring Boot creates self-contained, executable JARs, whereas traditional Spring requires a servlet container (e.g., Tomcat).
- **Embedded Servers**: Spring Boot provides embedded servers (Tomcat, Jetty), eliminating the need to deploy WAR files.

2. How do you create a Spring Boot application?

Answer:

To create a Spring Boot application:

1. **Create a Spring Boot Starter Project** using Spring Initializr (https://start.spring.io/).
2. **Add dependencies** (e.g., Spring Web, Spring Data JPA) in the pom.xml or build.gradle.
3. **Create the main class** annotated with @SpringBootApplication.

java

CODE

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

4. **Run the application**: Use mvn spring-boot:run or run the main class.

3. What is the role of application.properties or application.yml in Spring Boot?

Answer:

application.properties or application.yml are configuration files used to externalize application settings in Spring Boot. They allow you to define various properties such as:

- **Database settings** (URL, username, password).
- **Server settings** (port, context path).
- **Spring-specific settings** (logging level, JPA configurations).

Example of application.properties:

properties

CODE

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=secret
```

4. Explain the purpose of @SpringBootApplication and its components.

Answer:

@SpringBootApplication is a convenience annotation that combines three annotations:

1. **@EnableAutoConfiguration**: Enables Spring Boot's auto-configuration mechanism, which automatically configures components based on the application's dependencies.
2. **@ComponentScan**: Tells Spring to scan for components (beans) in the package where the main class is located.
3. **@Configuration**: Marks the class as a source of bean definitions.

5. How does Spring Boot handle dependency management?

Answer:

Spring Boot manages dependencies through **starters**, which are predefined sets of dependencies that support common features (e.g., spring-boot-starter-web, spring-boot-starter-data-jpa). Spring Boot uses a **parent POM** for consistent version management, eliminating version conflicts.

6. What is Spring Boot Actuator?

Answer:

Spring Boot Actuator provides built-in endpoints to monitor and manage the application in production. It exposes health checks, metrics, environment details, and more.

Common endpoints include:

- /actuator/health: Shows the health of the application.
- /actuator/metrics: Provides metrics about the application's performance.

7. Explain the process of creating a REST API in Spring Boot.

Answer:

To create a REST API in Spring Boot:

1. **Create a Spring Boot project** with spring-boot-starter-web dependency.
2. **Define a REST Controller** using @RestController annotation.
3. **Create RequestMapping methods** for CRUD operations using @GetMapping, @PostMapping, etc.

java

CODE

```
@RestController
@RequestMapping("/api")
public class MyController {
  @GetMapping("/hello")
  public String sayHello() {
    return "Hello, World!";
  }
}
```

4. **Run the application**: The API will be available at http://localhost:8080/api/hello.

8. How do you handle exceptions in Spring Boot?

Answer:

Spring Boot handles exceptions using @ControllerAdvice to globally handle exceptions or @ExceptionHandler to handle specific exceptions in a controller.

Example:

java

CODE

```
@ControllerAdvice
public class GlobalExceptionHandler {
  @ExceptionHandler(Exception.class)
  public ResponseEntity<String> handleException(Exception e) {
    return new ResponseEntity<>("Error: " + e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

9. What are Spring Boot DevTools, and how do they help in development?

Answer:

Spring Boot DevTools provides features to improve the development experience, such as:

1. **Automatic Restart**: Automatically restarts the application when code changes are detected.
2. **LiveReload**: Refreshes the browser when resources (e.g., HTML, CSS) change.
3. **Enhanced Logging**: Provides additional debugging information in the logs.

10. How does Spring Boot manage security?

Answer:

Spring Boot integrates with **Spring Security** to handle authentication, authorization, and other security features. It can be configured easily using application.properties or Java configuration classes.

Example for basic authentication:

properties

CODE

```
spring.security.user.name=admin
spring.security.user.password=secret
```

Spring Boot automatically configures default security settings, but these can be customized according to requirements.

This concludes the detailed answers for all the questions from the chapters. These answers are tailored for exam purposes and provide a concise yet comprehensive understanding of each topic.