Inventive Media

Java Programming Essentials - Day 4

Jansen Ang

What are we learning today?

- Annotations
- JDBC
- Creating Web Applications
  - Server-Side Programming
  - Spring Framework
  - Spring Boot
  - Thymeleaf

Annotations

What are Annotations?

Annotations are a form of metadata that provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate. Introduced in Java 5, annotations are used to give additional information to the compiler or to provide configuration and setup for various frameworks and libraries.

Why are Annotations Useful?

1. **Code Simplification**: Annotations can simplify the code by reducing boilerplate. For example, the @Override annotation clearly indicates a method is overriding a method from a superclass.
2. **Framework Integration**: They are heavily used in frameworks (like Spring, Hibernate) for defining configurations, which traditionally required extensive XML or property file settings.
3. **Compile-Time Checks**: Some annotations help in error detection at compile time. For instance, @Deprecated lets other developers know that a method is deprecated and should not be used.
4. **Runtime Processing**: Annotations can be used to affect the behavior of a program at runtime. Many Java frameworks use annotations for runtime processing.
5. **Code Readability**: They can make code easier to read and maintain by embedding configuration or behavior directly into the code.

Annotation Examples

Built-in Java Annotations:

- **@Override**: Indicates that a method overrides a method in a superclass.
- **@Deprecated:** Marks methods that should no longer be used.
- **@SuppressWarnings:** Instructs the compiler to suppress specific warnings.

Custom Annotations

```
@MyCustomAnnotation(value = "Example")
public class MyClass { ... }
```

```
public @interface MyCustomAnnotation {
    String value();
}
```

JDBC

Java Database Connectivity API

JDBC (Java Database Connectivity) is an API (Application Programming Interface) in Java for connecting and executing queries on a database. It provides a standard method for Java applications to interact with any SQL-compliant database.

Where is JDBC Used?

JDBC is widely used in Java applications that require interaction with a database, such as:

- **Web Applications**: For accessing and managing data stored in a database.
- **Enterprise Applications**: In systems like ERP or CRM, where data storage and retrieval are crucial.
- **Desktop Applications**: That require a database to store application data.
-

Core Components of JDBC

1. **DriverManager**: Manages a list of database drivers. It is used to establish a connection to the database.
2. **Connection**: Represents a connection with a specific database.
3. **Statement/PreparedStatement**: Used to execute SQL queries.
4. **ResultSet**: Represents the result set of a query.
5. **SQLException**: Handles SQL-related errors.

DriverManager

DriverManager is a class in Java's JDBC (Java Database Connectivity) API that manages a list of database drivers. It is responsible for establishing a connection between a Java application and a database.

Key Responsibilities of DriverManager:

1. **Loading Database Drivers**: It loads JDBC drivers for various databases. A JDBC driver is a set of Java classes that implement the JDBC interfaces to communicate with a specific type of database (like MySQL, Oracle, etc.).
2. **Establishing Connections**: It establishes a connection to a database using a database URL, username, and password provided by the user.
3. **Managing a Pool of Connections**: It can handle multiple connections simultaneously, each potentially to different databases.

Why You Need Drivers in JDBC

In JDBC (Java Database Connectivity), drivers are essential components that enable Java applications to interact with a database. Here's why they are needed:

1. **Database Communication**: JDBC drivers provide the necessary implementation for the JDBC interfaces to communicate with specific types of databases (like MySQL, Oracle, SQL Server, etc.). Each database has its unique way of handling queries and data, and the driver translates Java calls to the format understood by the database.
2. **Abstraction Layer**: Drivers abstract the underlying database specifics, allowing developers to interact with different databases in a uniform manner using JDBC APIs. This means you can switch databases with minimal changes in your Java code.
3. **Database-Specific Features:** They handle database-specific details like handling network communication, connection pooling, and transaction management.

Getting JDBC Drivers

1. Identify the JDBC Driver: Determine the JDBC driver for your database. For example, if you are using MySQL, the driver is typically mysql-connector-java.
2. Download the Driver:
    - Visit the official website of the database vendor.
    - Navigate to the downloads section and look for the JDBC driver.
    - For MySQL, you can download the driver from the MySQL Connector/J page.
3. Add the Driver to Your Project:
    - Once downloaded, extract the JAR file if it's in a compressed format.
    - In your Java project, create a folder named lib (if it doesn't already exist) and place the JAR file in this folder.
    - In your IDE (like Eclipse or IntelliJ IDEA), add the JAR file to your project's build path. This is usually done by right-clicking on the project → Properties → Java Build Path → Libraries → Add JARs (or Add External JARs).
4. Use the Driver in Your Application: After adding the driver to your classpath, you can use JDBC to connect to your database as usual.

Connection

In Java's JDBC (Java Database Connectivity) API, a Connection represents a session between a Java application and a database. It is one of the core components in JDBC, essential for establishing communication with a database.

Key Aspects of a JDBC **Connection**:

1. **Establishing a Session**: A **Connection** object is used to establish a link with the database, which is required for sending SQL statements and receiving results.
2. **Transaction Management**: It allows you to manage transactions (commit, rollback) and control transaction isolation levels.
3. **Creating Statements**: Through a **Connection**, you can create **Statement**, **PreparedStatement**, and **CallableStatement** objects to execute SQL queries.
4. **Database MetaData**: It provides metadata about the database, like its structure, version, and capabilities.
5. **Resource Management**: It’s crucial to close the **Connection** properly after use to free up database resources, which can be efficiently done using try-with-resources in Java.

Different Types of Statements

Statement

- Usage: Used for general-purpose access to the database, suitable for executing static SQL queries.
- Characteristics:
  - You can execute any SQL query using Statement.
  - It does not accept parameters

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM myTable");
```

PreparedStatement

- Usage: Extends Statement. Used for executing precompiled SQL statements. Ideal for queries that are executed multiple times.

Characteristics:

- Improves performance, especially for repeated executions, as the SQL statement is compiled only once.
- You can pass parameters to SQL queries using placeholders (?), making it more secure against SQL Injection attacks.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM myTable WHERE id = ?");
pstmt.setInt(1, 10); // Set the first parameter (1) to value 10
ResultSet rs = pstmt.executeQuery();
```

Different Types of Statements

CallableStatement

- Usage: Used to execute stored procedures in the database.

Characteristics:

- Can call stored procedures with or without parameters.
- Supports both input and output parameters.

```
CallableStatement cstmt = conn.prepareCall("{CALL myStoredProc(?)}");
cstmt.setInt(1, 10); // Set input parameter
ResultSet rs = cstmt.executeQuery();
```

Statement Best Practices

1. Use **PreparedStatement** for Most Operations: Due to its security and performance advantages, it's generally best to use PreparedStatement for executing SQL queries, especially when dealing with user input.
2. Use **Statement** for Static Queries: When you have a simple, one-time, static SQL query, Statement is sufficient.
3. Use **CallableStatement** for Stored Procedures: When you need to call stored procedures, especially those with input/output parameters, use CallableStatement.
4. **Close Resources:** Always close ResultSet, Statement/PreparedStatement/CallableStatement, and Connection objects to free up database resources. This can be done easily using try-with-resources statement in Java.

ResultSets

In Java's JDBC API, a ResultSet represents a table of data generated by executing a SQL query. It acts as a pointer to the results returned from a database query and provides methods to iterate through the rows and read data from columns.

Key Characteristics of ResultSet:

1. **Cursor Movement**: The ResultSet cursor, which points to the current row, can be moved forward (and in some cases, backward or to a specific row) to navigate through the rows.
2. **Data Retrieval**: It allows retrieval of data from each column in the current row using various getter methods like getString, getInt, getDate, etc.
3. **Concurrency and Type**: The type of a ResultSet object (scrollable or not, updateable or not) depends on the arguments passed when creating a Statement or PreparedStatement.
4. **Lifecycle**: After the Statement or PreparedStatement that generated it is closed, or the Connection is closed, the ResultSet becomes invalid.

Parameter Data Binding

Parameter data binding in JDBC refers to the process of binding values to placeholders (parameters) in a SQL statement, typically a PreparedStatement. This feature enhances the security and efficiency of executing SQL statements.

Advantages of Parameter Data Binding

1. Prevention of SQL Injection: By using parameterized queries, you can protect your application against SQL injection attacks.
2. Improved Performance: Precompiling the SQL statement with placeholders can improve performance, especially when the statement is executed multiple times.
3. Code Readability and Maintenance: It makes the code more readable and easier to maintain.

```
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
 // other code...
PreparedStatement pstmt = conn.prepareStatement(sql)) {

// Binding parameters
pstmt.setString(1, name);  // Bind 'name' to the first placeholder
pstmt.setString(2, email); // Bind 'email' to the second placeholder

// Executing the statement
int affectedRows = pstmt.executeUpdate();
```

Creating Web Applications

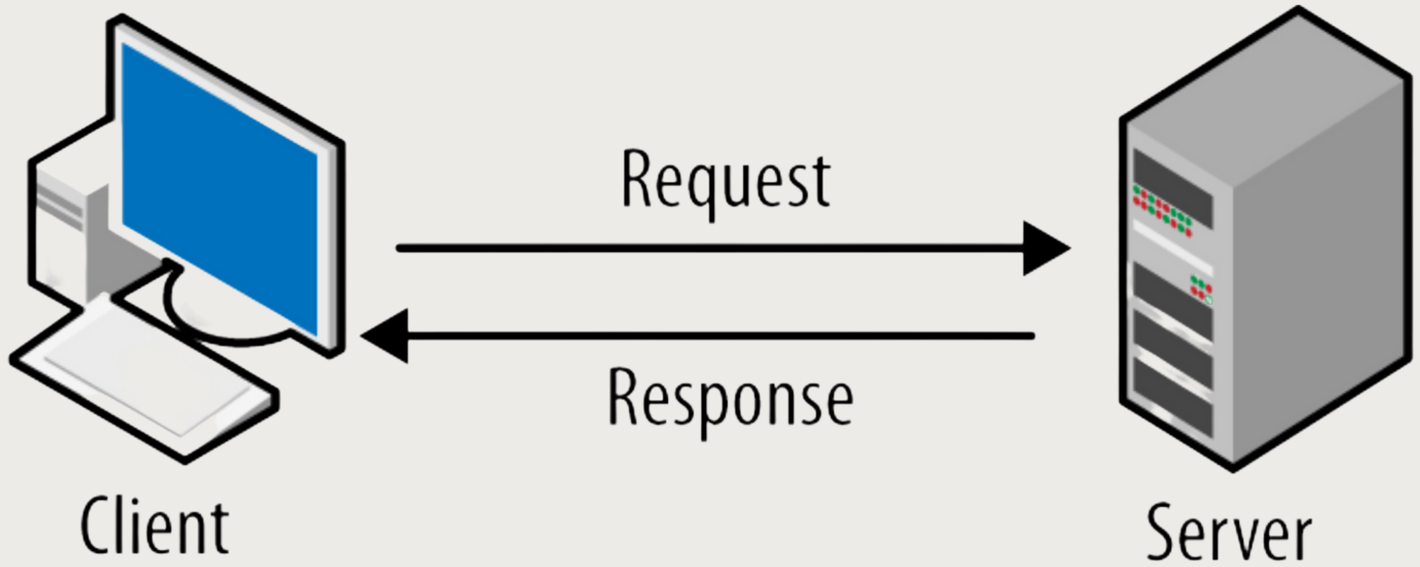
How Server-Side Programming works

Client vs. Server Side Programming

- **Client-Side**
  - Description: Code executed on the user's browser.
  - Languages: HTML, CSS, JavaScript.
  - Purpose: Enhancing user interface and experience, immediate interaction without server calls.
  - Example: Validating a form input on a webpage before submission.
- **Server-Side**
  - Description: Code executed on the web server.
  - Languages: Java (Servlets, Spring), PHP, Python (Django, Flask), Ruby on Rails, etc.
  - Purpose: Handling business logic, database operations, authentication.
  - Example: Retrieving user data from a database after a login form is submitted.

How Server-Side Programming works

- The server receives a request from a client (usually a web browser).
- The server-side script processes the request. This might involve querying a database, processing data, or other operations.
- The server sends a response back to the client. This could be a full HTML page, JSON/XML data, etc.



Java’s Role in Web Development

- **Server-Side Programming**: Java is predominantly used for server-side programming. It enables the creation of robust, scalable, and secure web applications.
- **Enterprise Solutions**: Widely used in enterprise environments for its stability and scalability.
- **Frameworks and APIs**: Rich set of frameworks and APIs for developing web applications, such as Spring, Hibernate, etc.

Java’s Role in Web Development

Servlets: The Building Blocks

Servlets are Java's answer to handling HTTP requests and responses in a web application. They act as a middle layer between requests coming from a web browser or other HTTP clients and databases or applications on the HTTP server.

JavaServer Pages (JSP)

JSPs are file with .jsp extension that are used for building dynamic web content. They allow embedding Java code in HTML pages, which makes them ideal for creating the view layer of a web application.

Templating Engines

While JSP is traditional for Java web applications, modern applications often use templating engines like Thymeleaf.

Thymeleaf: It's a modern server-side Java templating engine, used for rendering HTML on the server. Thymeleaf has several advantages over JSP:

- **Natural Templating:** Thymeleaf’s templates are natural; they can be displayed in browsers and work as static prototypes.
- **Integration with Spring:** Thymeleaf offers seamless integration with Spring Framework, making it a preferred choice for Spring-based applications.
- **Syntax and Features:** Thymeleaf provides a rich set of attributes for conditional rendering, loops, and template layouts, which enhances the capability to create dynamic views.

Build Tools

Build Tools

Build tools are software tools designed to automate the creation of executable applications from source code. They handle tasks like compiling code, packaging binary code, running tests, and deployment.

Why are Build Tools Useful?

1. **Automation**: Automates repetitive tasks like compiling, linking, and packaging the code, which can be error-prone if done manually.
2. **Consistency**: Ensures consistency in the build process across different environments and among different team members.
3. **Dependency Management**: Manages libraries and dependencies the application needs, automatically downloading and integrating them into the build process.
4. **Efficiency**: Increases development efficiency by reducing manual overhead and integrating with Continuous Integration/Continuous Deployment (CI/CD) pipelines.
5. **Testing and Integration**: Facilitates testing and integration by running tests automatically during the build process.

Ant

- Description: Apache Ant is a Java-based build tool. It uses XML to describe the build process and its dependencies.
- Features:
  - Less convention-heavy compared to Maven.
  - Highly customizable with the use of XML.
- Use Case: Suitable for Java projects where fine-grained build process control is needed.

Build Tools

Maven

- Description: Apache Maven is a powerful build automation tool used primarily for Java projects. It uses an XML file (pom.xml) to manage project dependencies and build configuration.
- Features:
  - Strong dependency management.
  - Plugin-based architecture.
  - Project Object Model (POM) for configuration.
- Use Case: Ideal for complex Java projects with extensive dependencies.

Gradle

- **Description**: Gradle is an open-source build automation tool that builds upon the concepts of Apache Ant and Maven but introduces a Groovy-based domain-specific language (DSL) for describing builds.
- **Features**:
  - Combines the best features of Ant and Maven.
  - Supports incremental builds.
  - Flexibility and scalability.
- **Use Case**: Preferred in Android app development and projects requiring high customizability.

Maven

Apache Maven is a popular build automation and project management tool used primarily in Java projects. It streamlines and simplifies the build process for Java applications.

Why is Maven Used?

- **Dependency Management**: Maven automates the process of downloading dependencies (libraries and frameworks) required for a project from a central repository, managing them in a standardized way.
- **Standardized Build Process**: It provides a uniform build system, so developers can quickly start and work on new projects without having to handle project build specifics.
- **Project Lifecycle Management**: Maven uses a predefined lifecycle to manage the build process, including steps like compilation, testing, packaging, and deployment.
- **Plugin-based Architecture**: Maven extends its capabilities through plugins, allowing it to support various languages and technologies beyond Java.
- **Reproducibility**: Ensures that the build process is consistent and reproducible across different environments and systems.

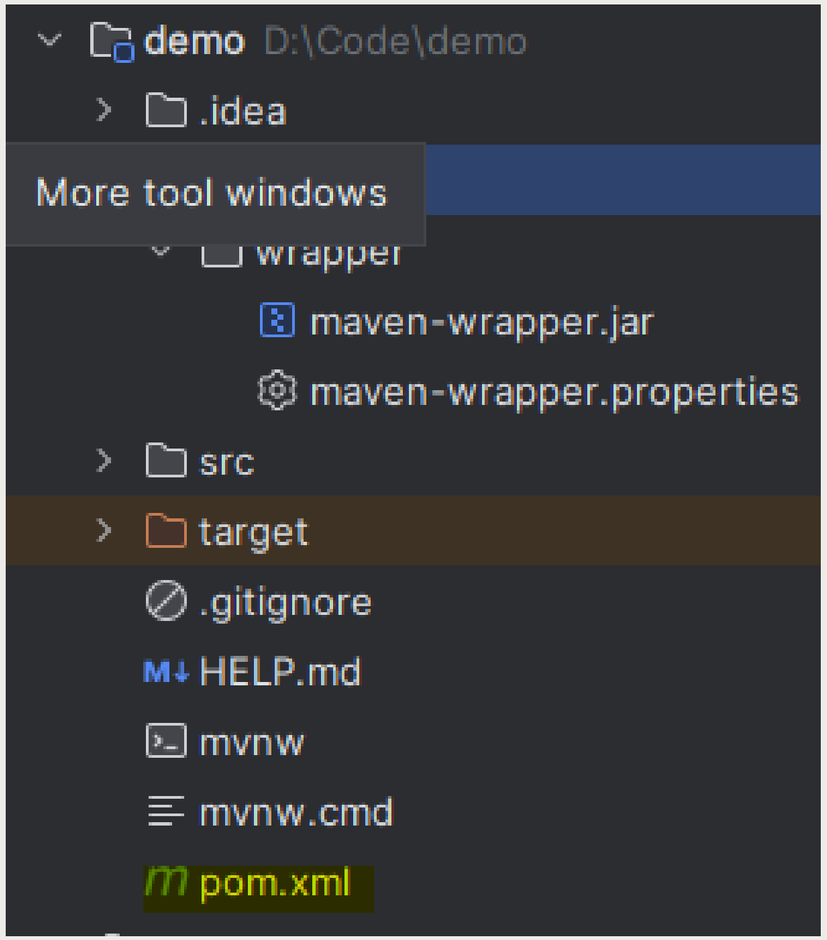
How is Maven Used?

- **Project Creation**: Maven can generate project skeletons, setting up a basic structure automatically.
- **Building and Testing**: It compiles source code, runs tests, and packages binary code into deployable formats like JARs and WARs.
- **Managing Dependencies**: Automatically handles the downloading and updating of dependencies.

The POM File and its Structure

- The core of a Maven project is the pom.xml file (Project Object Model). It defines the project configuration, including the following key elements:
- **<modelVersion>**: The version of the POM model used (usually 4.0.0).
- **<groupId>**: The unique identifier of your organization or group.
- **<artifactId>**: The unique identifier of the project.
- **<version>**: The version of the project.
- **<dependencies>**: A list of project dependencies, each specified with groupId, artifactId, and version.
- **<build>**: Contains build configurations and plugin settings.
- **<properties>**: Project-wide properties like Java version.

The POM File and its Structure




```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
         maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId>
    <artifactId>my-application</artifactId>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.3</version>
        </dependency>
        <!-- Other dependencies -->
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <!-- Plugin configuration -->
            </plugin>
            <!-- Other plugins -->
        </plugins>
    </build>
</project>
```

Spring Framework

What is the Spring Framework?

- The Spring Framework is an open-source application framework and inversion of control container for the Java platform. It is one of the most popular frameworks for building Java applications, particularly enterprise-level applications.

History of Spring Framework

- **Origins**: Spring was first released in 2003 by Rod Johnson as an alternative to the complex Java EE (Enterprise Edition) at the time. It was designed to simplify enterprise Java development and address the difficulties presented by earlier J2EE specifications.
- **Evolution**: Over the years, Spring has evolved significantly, adding numerous features and modules to support a wide range of functionalities, including data access, transaction management, web applications, RESTful services, and more.
- **Spring Boot**: In 2014, Spring Boot was introduced, offering a way to easily create stand-alone, production-grade Spring-based applications that can be "just run."

Why is Spring Framework Used?

- **Simplification of Enterprise Java:** Spring makes it easier to develop enterprise Java applications by reducing the need for complex EJB (Enterprise JavaBeans) and providing a simpler programming model.
- **Dependency Injection**: One of the core features of Spring is Dependency Injection (DI), which helps in creating loosely coupled applications, thereby improving maintainability and testability.
- **Aspect-Oriented Programming:** Spring supports Aspect-Oriented Programming (AOP), allowing for the separation of cross-cutting concerns (like logging, security, etc.) from the business logic.
- **Modular Design:** Spring is modular, meaning developers can pick and choose which modules are necessary for their application, leading to lightweight applications.
- **Integration Capabilities:** It offers excellent integration with other frameworks and technologies, making it a versatile choice for various types of applications.
- **Robust Ecosystem:** The Spring ecosystem includes projects like Spring Boot, Spring Security, Spring Data, and more, providing a comprehensive suite of tools for different needs.
- **Community and Support:** Spring has a large and active community, providing a wealth of resources, extensive documentation, and support.

Usage of Spring Framework

- **Web Applications**: With Spring MVC (Model-View-Controller), developers can create web applications with flexible routing and view templating.
- **RESTful Web Services:** Spring's support for RESTful services allows for the creation of APIs for modern web applications.
- **Data Access:** Spring simplifies interactions with databases through Spring Data, JDBC, and ORM (Object-Relational Mapping) integration.
- **Transaction Management:** Offers a consistent transaction management interface that scales down to local transactions and scales up to global transactions (JTA).
- **Security:** Spring Security provides comprehensive security services for Java EE-based enterprise software applications.

The Spring Container

The Spring Container is the core component of the Spring Framework. It creates, manages, wires, and destroys the beans (objects) through a process known as Dependency Injection. The Spring Container is responsible for instantiating, configuring, and assembling objects known as beans, which represent the application's business logic and are defined in the Spring configuration files.

Types of Spring Containers

There are two primary types of Spring Containers:

1. **BeanFactory**: The simplest container, providing basic support for DI. It's lightweight and is used for simple scenarios.
2. **ApplicationContext**: A more advanced container that offers all **BeanFactory** capabilities plus more enterprise-specific features like event propagation, declarative mechanisms, and AOP integrations. It is generally recommended for most use cases.

Inversion of Control (IoC)

IoC is a principle in software engineering where control over the flow of a program is transferred from the program itself to a framework or container.

- IoC is achieved through DI. The Spring Container injects dependencies into the components (beans) instead of them creating or looking up their dependencies.
- Advantages:
  - Reduces the boilerplate code necessary for initializing dependencies.
  - Promotes loose coupling and increases flexibility and maintainability of the code.

Dependency Injection

Dependency Injection is a design pattern used by the Spring Framework to implement Inversion of Control (IoC) for better decoupling of code.

- **Concept**: Instead of objects creating dependencies themselves, they are injected by an external entity (i.e., the Spring Container).
- **Types of DI in Spring:**
  - Constructor Injection: Dependencies are provided through a class constructor.
  - Setter Injection: The container calls setter methods on your beans after invoking a no-argument constructor or a no-argument static factory method to instantiate your bean.
- **Benefits**:
  - Simplifies unit testing.
  - Increases code reusability.
  - Improves code decoupling.

Spring Beans

In the Spring Framework, a "bean" is a fundamental concept. It's an object that is instantiated, assembled, and otherwise managed by a Spring IoC (Inversion of Control) container. Beans are the backbone of any Spring application.

How Spring Finds Beans

1. **Component Scanning**: Spring automatically detects and registers beans by scanning the application classes for annotations like @Component, @Service, @Repository, and @Controller. This is done during the application startup.
2. **XML Configuration:** In traditional Spring applications, beans can also be defined in an XML configuration file.
3. **Java-based Configuration**: Beans can be declared in Java configuration classes using @Bean annotation. This approach is part of Java-based configuration in Spring.
    -

Spring Beans

How You Use Them

- **Dependency Injection:** Beans can be injected into other beans or components. This is typically done via constructor or setter injection using annotations like @Autowired.
- **Defining Business Logic**: Beans usually contain the business logic of your application. For example, a @Service annotated class would contain service-level business logic.
- **Data Access Objects (DAO):** Beans can be used to define data access objects to interact with the database.
- **Controllers**: In a Spring MVC application, controllers are defined as beans that handle HTTP requests.

Why They Are Useful

1. **Decoupling**: Spring beans promote loose coupling through dependency injection. This makes the application easier to test and maintain.
2. **Lifecycle Management:** The Spring container manages the entire lifecycle of beans.
3. **Singletons by Default:** By default, Spring manages beans as singletons within the application context, ensuring there is only one instance of a bean per container. This can be changed based on the required scope.
4. **AOP Support:** Beans can be integrated with Spring's Aspect-Oriented Programming (AOP) features to add behavior (like logging, transaction management) without modifying the business logic.
5. **Efficient Resource Management:** Spring handles the creation and management of beans efficiently, which can optimize memory and resource usage.

Spring Boot

What is Spring Boot?

- Spring Boot is an extension of the Spring framework that simplifies the initial setup and development of new Spring applications. It's designed to minimize the amount of configuration and setup required to get a Spring application up and running.

Key Features of Spring Boot

- **Auto-Configuration**: Spring Boot automatically configures your application based on the libraries present on the classpath. This simplifies the development process as it reduces the need for specifying beans and configurations.
- **Standalone**: Spring Boot applications are standalone and can be run independently of an application server. They come with an embedded web server (like Tomcat, Jetty, or Undertow).
- **Opinionated Defaults**: It provides 'starter' dependencies to simplify build and application configuration with sensible defaults.
- **Production-ready**: Includes features like health checks and metrics, which makes it easy to deploy and manage production applications.
- **No Code Generation and XML Configuration**: Requires no XML configuration or code generation, making the codebase cleaner and more maintainable.

How Does Spring Boot Differ from Spring Framework?

- **Configuration**: Traditional Spring framework requires a lot of XML configuration, whereas Spring Boot reduces this necessity through auto-configuration and convention over configuration principles.
- **Ease of Use**: Spring Boot is designed to get a Spring application up and running as quickly as possible, with minimal fuss. It simplifies dependency management by providing a set of starter POMs (Project Object Model).
- **Embedded Server**: Unlike Spring, where you need to deploy your application on a web server, Spring Boot comes with embedded web servers like Tomcat or Jetty, making deployment easier.
- **Microservices Ready:** Spring Boot is well-suited for microservice architectures due to its embedded server capability and easy deployment.
- **Rapid Prototyping**: With Spring Boot, you can quickly prototype and develop applications, making it ideal for agile development and fast-paced environments.

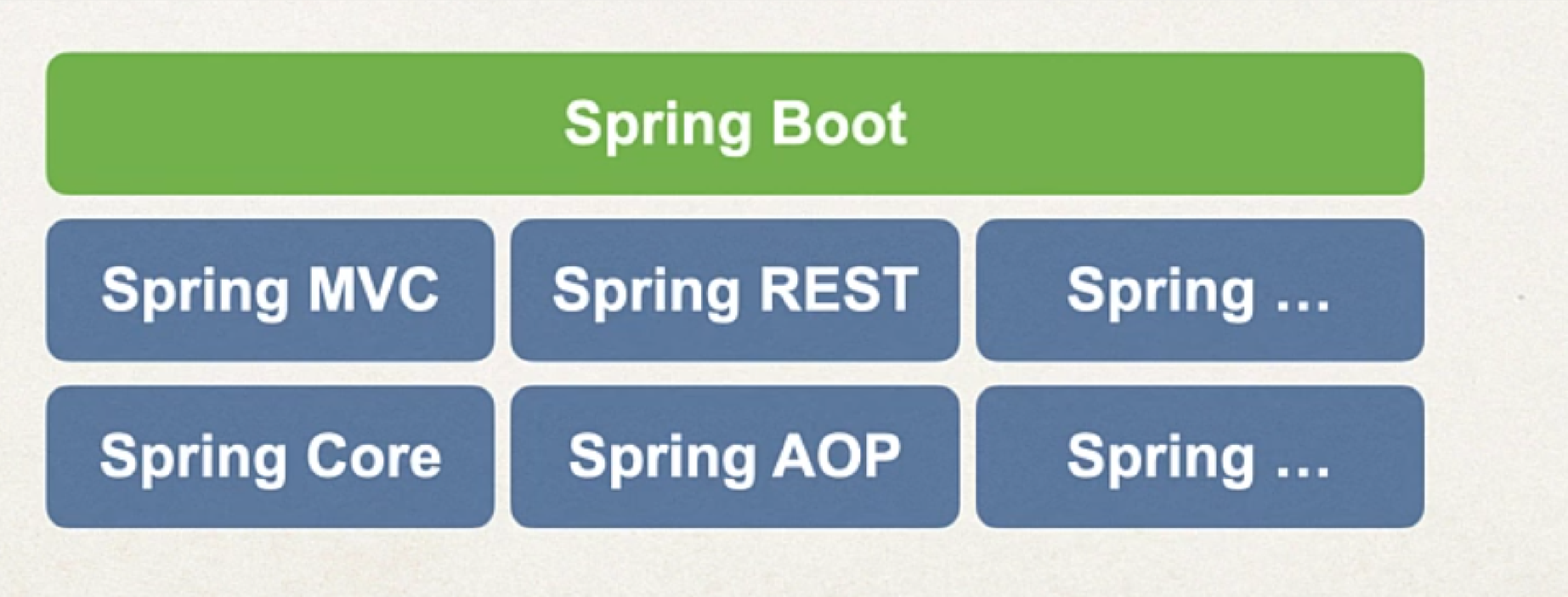
Spring Boot @SpringBootApplication

- **`@SpringBootApplication`** is composed of the following annotations:

Annotation	Description
`@EnableAutoConfiguration`	Enables Spring Boot's auto-configuration support
`@ComponentScan`	Enables component scanning of current package Also recursively scans sub-packages
`@Configuration`	Able to register extra beans with `@Bean` or import other configuration classes

Does Spring Boot replace Spring MVC, Spring Core, etc.?

No. Instead, Spring Boot actually uses those technologies.



Model class

The Model in Spring serves as a container for data that a controller wants to pass to the view layer.

Key Functions of Model in Spring

1. **Data Transfer**: The primary function of the Model is to transfer data from the controller to the view. This data typically represents the dynamic content of the web page.
2. **Attribute Storage**: It stores attributes, which can be any objects, and makes them accessible to the view. Controllers add data to the Model, and views access this data to render content.
3. **Abstraction from View**: The Model provides an abstract way of transmitting data without needing to know the details of the view layer. This abstraction facilitates the decoupling of the controller and view layers.

Spring Boot Annotations

1. **@Component**: Indicates that a class is a Spring component.
2. **@Autowired**: Marks a constructor, field, or setter method to be autowired by Spring's dependency injection facilities.
3. **@ComponentScan**: Configures component scanning directives for use with @Configuration classes.
4. **@Controller**: Marks a class as a web controller, capable of handling HTTP requests.
5. **@RestController**: A specialized version of **@Controller** that assumes every method returns a domain object instead of a view. It's a shortcut for **@Controller** and **@ResponseBody** combined.
6. **@RequestMapping**: Annotation for mapping web requests to specific handler methods. It can be used at the class or method level.
7. **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, **@PatchMapping**: Specialized shortcuts for **@RequestMapping** for different HTTP methods.
8. **@RequestParam**: Indicates a method parameter should be bound to a web request parameter.
9. **@PathVariable**: Indicates that a method parameter should be bound to a URI template variable.
10. **@RequestBody**: Indicates a method parameter should be bound to the body of the web request.
11. **@ResponseBody**: Indicates that a method's return value should be bound to the response body.
12. **@ResponseStatus**: Marks a method or exception class with the status **code()** and **reason()** that should be returned.

Thymeleaf

What is Thymeleaf?

- Thymeleaf is a modern server-side Java template engine used for rendering web pages, typically in web applications using Spring Framework. It's an open-source project and a popular choice for generating HTML dynamically.

Key Features of Thymeleaf

- **Natural Templating**: Thymeleaf's templates are readable and can be viewed in browsers as static files, even before they are rendered by the server, which makes them natural templates.
- **Integration with Spring**: Thymeleaf has excellent integration with Spring Framework, which makes it a preferred choice for web applications built with Spring.
- **Standard and Customizable Syntax**: Thymeleaf uses standard HTML syntax and also allows defining custom tags and attributes. This feature ensures that templates are both functional and maintainable.
- **Locale Support and Internationalization**: It supports text internationalization (i18n) for multi-language applications.
- **Expression Language**: Thymeleaf includes a powerful expression language for navigating data in the model and integrating logic into the template.

Thymeleaf Operations

Thymeleaf supports various types of expressions:

- **Variable Expressions**: ${...} for accessing data in the model.
- **Selection Variable Expressions**: *{...} for beans or forms.
- **Message Expressions**: #{...} for internationalization.
- **Link URL Expressions**: @{...} for creating URLs.
- **Literal Expressions**: '' for static text.

If condition

```
<div th:if="${someCondition}">
    This content will only be displayed if 'someCondition' is true.
</div>
```

```
<ul>
    <li th:each="item : ${items}" th:text="${item}">
        Item description
    </li>
</ul>
```

Any questions?

Jansen Ang