**웹개발 기초**

웹개발은 브라우저와 서버 사이의 요청과 응답을 잇는다. 따라서 네트워크와 브라우저, 서버를 비롯한 컴퓨팅 환경 전반에 대한 이해가 필요하다. 개인이 브라우저와 네트워킹, 서버 개발 환경에 관한 지식을 하나부터 열까지 익히고 바닥부터 쌓아 올리는 일은 불가능에 가깝다. 그렇기 때문에 현대 웹 개발은 많은 지식이 축적, 추상화된 프로토콜, 툴, 프레임워크를 활용해 이뤄진다.

**웹 애플리케이션 서버(WAS)와 웹 서버**

웹 서버는 WAS와 웹 서버로 구분된다. 웹 애플리케이션 서버에는 서버 사이드 프로그램이라고 불리는 방식의 코드들이 동작한다. 클라이언트가 특정 데이터에 대한 동적인 반응을 원하면 WAS 서버는 해당 요청을 서버 사이드에 위치한 프로그램을 실행시켜 동적으로 변환한 후 응답한다. 자바는 서블릿과 JSP를 사용해 WAS 서버를 구성한다. 일반적인 웹 서버는 HTML, CSS, 이미지 파일 등의 변환이 불필요한 정적 파일을 보관하고 있다가 요청이 오면 전달한다.

자바는 이러한 웹개발 환경에 대응해 JavaEE라는 기술 스펙을 제공한다. 동시 요청 문제, 서버 장애, 데이터 전송 최적화, 분산 처리 등의 문제를 해결하기 위해 등장한 이 기술에서 일반적으로 가장 기본적인 기술이 서블릿과 JSP이다.

**서블릿**

서버에서 동적인 요청을 처리할 때 서블릿을 사용한다. 서블릿 클래스라고 불리는 클래스들은 고객의 요청에 대응하기 위한 데이터와 메서드를 가지고 있다. 이러한 서블릿 클래스는 해당 서블릿 클래스를 실행시켜줄 수 있는 프로그램을 통해 API로 기능한다. 이때 해당 서블릿 클래스의 코드를 클라이언트의 요청에 따라 실행시켜주는 프로그램이 톰캣이다. 톰캣과 같은 프로그램은 서블릿 컨테이너 또는 서블릿 엔진으로 불린다. 서블릿 컨테이너는 고객의 요청과 웹 애플리케이션 사이에서 서블릿 클래스나 JSP 파일의 실행을 담당한다. JSP는 근본적으로는 서블릿과 비슷하지만 좀 더 HTML에 특화된 방식으로 코드를 작성할 수 있도록 포매팅되어 있다. 이는 용도에 의한 구분으로 서블릿으로는 애플리케이션 로직을 처리하고 JSP로는 화면을 개발하기 위함이다.

일반적으로 자바 코드로 작성된 프로그램은 메인 함수를 통해 실행된다. 그러나 서블릿 클래스, JSP는 서블릿 컨테이너를 통해 실행되기 때문에 일반적인 자바 프로그램과 다음과 같은 차이가 생긴다.

1. 사용자가 아닌 컨테이너가 객체를 생성하거나 호출한다.
2. 서블릿 클래스에서 생성한 객체를 컨테이너가 관리한다.
3. 서블릿/JSP 코드의 개발은 기본적인 자바 API와 더불어 서블릿 API도 같이 사용해야 한다.

**JSP**

Java Server Page의 약자이다. HTML이 메인이고 자바 코드가 서브인 자바코드이다. JSP도 결국에는 서블릿 코드로 변환되어 컴파일되고 실행되기 때문에 자바 코드이다. 필요한 순간에 자바 파일로 생성되고 이를 컴파일 해 클래스 파일로도 만들어진다. 개발자의 목적과 편의성에 따라 순수 서블릿 코드와 JSP로 구분된다. JSP는 GET/POST 방식의 호출을 구분하지 않기 때문에 POST 방식으로 접근해야 하는 JSP도 GET 방식으로 호출할 수 있다는 문제가 있다. 또 JSP를 다른 JSP파일로 변경할 경우 주소의 변경 사항이 모든 액션 코드에 반영되어야 해서 유지 보수의 문제도 있다. 따라서 최근 JSP는 다음과 같이 제한적인 용도로 사용한다.

1. 쿼리 스트링이나 파라미터를 처리하지 않는다. – JSP 대신 서블릿을 통해 처리한다.
2. JSP는 입력 화면을 구성하거나 처리 결과를 보여주는 용도로만 사용한다.
3. 브라우저는 직접 JSP 경로를 호출하지 않고 서블릿 경로를 통해 JSP를 보는 방식으로만 사용한다.

위와 같은 문제를 해결하기 위해 나온 방식이 웹 MVC 방식이다. JSP는 결과만 출력하고 처리는 서블릿을 이용하는 방식으로 변화한 것이다.

**Web MVC 방식**

JSP는 GET/POST 구분 불가, URL 변경에 유연하지 못하다는 등의 단점이 있었다. 2000년대 중반부터 이런 단점을 극복하기 위해 MVC 패턴을 활용한 개발이 일반화되었다.

MVC는 위 문제를 해결하기 위해 다음과 같은 구조를 활용한다.

* **브라우저의 요청**
* **서블릿(컨트롤러)이 필요한 데이터(를 제공하는 객체 -> 모델) 처리 후 JSP로 전달**
* **JSP(뷰)는 EL을 이용해 최종 데이터 출력**

JSP를 통해 생성한 화면은 톰캣이 브라우저로 전송한다.

**PRG 패턴(Post-Redirect-GET)**

포스트와 리다이렉트를 결합해 활용하는 패턴이다. 사용자가 컨트롤러에 원하는 작업을 POST로 처리하기를 요청하면 POST를 컨트롤러에서 처리하고 브라우저는 다른 경로로 이동(GET)하라는 응답(리다이렉트)를 주게 된다. 브라우저는 GET 방식으로 이동하게 된다. 개발자 도구에서 응답 헤더의 location 경로에 해당한다.

**HttpServlet**

HttpServlet은 Http 메서드를 처리하는 메서드인 doGet, doPost등의 메서드를 제공한다. 개발자들은 이 메서드를 오버라이트 해서 각 메서드에 대한 반응을 작성할 수 있다. HttpServlet을 상속한 클래스는 톰캣과 같은 WAS의 내부에서 자동으로 객체를 생성하고 관리해 개발자가 신경 쓸 필요가 없다. HttpServlet은 또한 멀티 스레드에 의해 동시에 실행될 수 있도록 개발되기 때문에 개발자가 동시 요청에 대한 부담을 덜 수 있다.

HttpServlet은 Servlet, Serializable, ServletConfig을 구현하는 GenericServlet이라는 추상 클래스를 상속해 구현된다. 제네릭한 서블릿은 HttpRequest와 HttpResponse 타입이 나닌 일반적인 서블릿리퀘스트, 서블릿리스폰스라는 타입의 매개변수를 인자로 받는다.

라이프 사이클은 단순화하면 요청을 처리해 응답하는 과정이다. 이 과정에 톰캣이 개입해 객체의 생성과 관리를 돕는다. 브라우저가 특정 경로를 호출하면 톰캣은 서블릿에서 이를 처리해 해당 서블릿 클래스를 로딩하고 객체를 생성한다. 이때 init()이라는 메소드를 실행해 서블릿 객체가 동작하기 전에 수행할 일들을 처리할 수 있다. 서블릿 객체는 브라우저의 요청에 대한 정보를 분석해 GET/POST 등의 정보와 파라미터(쿼리 스트링의 내용)를 HttpServletRequest의 파라미터로 전달받는다. 서블릿 내부에서 doGet(), doPost() 등의 메소드를 실행하고 동일한 호출이 있을 때는 동일한 객체 하나만을 사용해 이를 처리하게 된다. 톰캣이 종료될 때는 서블릿의 destroy()메서드를 실행한다.

서블릿 객체는 경로에 맞게 하나만 생성되고 호출시에는 자동으로 doGet이나 doPost 등을 호출한다.

**모델**

모델은 컨트롤러를 담당하는 서블릿과 뷰를 담당하는 JSP를 제외한 나머지 부분이다. 크게 서비스(비즈니스 로직 처리) 계층, 영속(데이터 처리) 계층으로 분리된다.

**DTO(Data Transfer Object)**

3티어와 같이 계층을 분리하는 경우 반드시 계층이나 객체들 간에 데이터 교환이 이루어진다. 한 개 이상의 데이터를 전달할 경우가 많기에 묶어서 하나의 객체로 전달하게 되는데 이 객체를 DTO(Data Transfer Object)라고 한다.

특별한 규약이나 제약은 없지만 대부분 Java Beans 형태로 구성하는 경우가 많다. Java Beans는 다음과 같은 형식으로 구성된다.

* 생성자가 없거나 반드시 파라미터가 없는 생성자 함수를 가지는 형태
* 멤버 변수는 private로 작성
* Getter/setter의 제공

**EL(Expression Language)**

표현 언어는 ${}와 같은 형태로 사용한다. 표현 언어 안에 들어간 개체는 자동으로 getter를 호출하게 된다 ex. ${list[0].title}의 형태로 사용하면 list[0]의 title을 출력하게 된다. DTO 클래스는 자바 빈즈 형태이기에 멤버 변수가 private로 작성되어 있지만 EL은 접근제한자와 관계없이 멤버 변수를 출력할 수 있다.

**JSTL(JavaServer Pages Standard Tag Library)**

JSP에서 동작하는 새 태그들의 묶음이다. 자바 문법보다 간결하게 제어, 반복, 선언문등을 처리할 수 있고 확장 가능하다. 라이브러리가 필요하기 때문에 빌드 파일에 의존성 라이브러리를 추가해 사용한다. c:foreach, c:if test, c:choose c:when…

**MVC의 모델**

정리하자면 서블릿은 Http메서드를 컨트롤하고 JSP는 뷰를 담당하고 모델은 DTO(데이터)와 DTO를 가공하는 서비스 로직(서비스)로 구분

**JDBC(Java Database Connectivity)**

자바 프로그램과 데이터베이스를 연결해 데이터를 교환하는 프로그램이다. 이때 네트워크 데이터를 처리하는 코드가 필요한데 이를 처리해주는 역할을 하는 게 JDBC 드라이버의 API이다.

JDBC의 흐름

1. 네트워크를 통해 데이터베이스와 자바 프로그램이 연결을 맺는다.
2. 자바를 통해 SQL을 작성하고 데이터베이스로 전송한다.
3. 필요할 경우 데이터 베이스에서 중간 결과를 받아 자바에서 재처리한다.
4. 데이터베이스와 연결을 종료한다.

JDBC를 활용하기 위해서는 몇 가지 필수적인 용어들과 API를 숙지하는 게 좋다.

**java.sql.Connection**

Connection 인터페이스는 데이터베이스와 네트워크상의 연결을 의미한다. 데이터베이스에서 SQL을 실행하기 위해 반드시 Connection 타입의 객체를 생성해야 한다. Connection 인터페이스를 활용해 실제 구현 클래스는 JDBC 드라이버 파일 내부의 클래스를 이용한다. 가장 중요한 것은 Connection은 반드시 close()해야 한다는 사실이다. 연결이 종료되지 않으면 새 연결을 받을 수 없는 상황이 발생하기 때문이다. Close()는 데이터베이스 쪽에 연결을 끊어도 좋다는 신호를 주고 네트워크 연결을 종료시킨다.

Connection의 종료는 일반적으로 트라이 캐치 파이널리 문이나 트라이 윗 리소스 방식을 이용한다. Connection 클래스의 가장 중요한 기능은 Statement, Prepared-Statement와 같이 SQL을 실행할 수 있는 객체를 생성하는 기능이다.

**java.sql.Statement/PreparedStatement**

전자는 쿼리와 데이터를 함께 전달하고 후자는 쿼리만 우선 전달하고 이후에 데이터를 전송한다. 실무에서는 후자만 사용하는데 SQL injection을 막기 위해서이다. 세터, executeUpdate()[DML 실행 후 결과를 int 타입으로 반환, 결과는 몇 개의 행이 영향을 받았는가이다.], executeQuery()[쿼리 실행 시 사용] 등의 기능이 있다. Connection과 마찬가지로 close()를 통해 종료해야 데이터베이스 내부에서 메모리 등 사용한 자원들이 즉각적으로 정리된다.

**java.sql.ResultSet**

insert, update, delete 등의 DML은 위 API를 사용(몇 행에 반영되는지만 리턴하므로)하지만 select가 반환하는 데이터를 읽기 위해서는 ResultSet이라는 인터페이스를 이용한다. 자바코드에서 데이터를 읽기 때문에 게터를 이용해서 필요한 타입으로 데이터를 읽는다. next()[다음행 읽도록 이동], close() 가 있다.

**Connection Pool과 DataSource**

JDBC는 필요한 경우에만 순간적으로 연결을 설정하고 필요하지 않은 경우 연결을 해제한다. 이는 계속해서 커넥션을 유지하는 데 많은 리소스가 할당되기 때문이다. SQL을 많이 사용할 수록 오버헤드가 증가하기 때문에 커넥션 풀을 이용해 이 문제를 해결한다.

커넥션 풀은 미리 커넥션을 생성해 보관하고 필요할 때마다 커넥션을 꺼내서 쓰는 방식이다.

**매번 만들어 쓰는 것과 커넥션 풀을 만들어 쓰는 것의 차이**

In JDBC, the concept of creating a new connection versus using a connection pool involves significant differences in terms of performance, resource utilization, and scalability. Understanding these differences is crucial in developing efficient and robust database-driven applications.

### Creating a New Connection

- \*\*Process\*\*: Every time you create a new connection, the application must establish a fresh connection with the database server. This involves network communication, authentication, transaction setup, and various other initialization tasks.

- \*\*Cost\*\*: Establishing a new database connection is resource-intensive and time-consuming, particularly because of the overhead of network communication and the initialization process on both the client and server sides.

- \*\*Scalability\*\*: If an application creates and closes connections frequently (especially under heavy load), the overhead can significantly impact performance and scalability. It can lead to a higher load on the database server and increased latency in the application.

### Using a Connection Pool

- \*\*Process\*\*: A connection pool maintains a pool of active database connections that are ready to be used. When an application needs to interact with the database, it borrows a connection from the pool, uses it, and then returns it to the pool. The connections in the pool are already established and authenticated.

- \*\*Performance\*\*: Because the connections are pre-established, the application avoids the overhead of setting up a new connection each time. Borrowing and returning connections to the pool is much faster than opening and closing them.

- \*\*Resource Utilization\*\*: Connection pools effectively reduce the number of open connections to the database, which minimizes the resource utilization on the database server. This is more efficient than having each client application instance create its own connections.

- \*\*Scalability and Reliability\*\*: Connection pooling enhances scalability by managing a finite number of connections that can be shared among many client requests. It also adds reliability by automatically handling failed connections (e.g., by removing broken connections and creating new ones).

### Summary

- \*\*Creating a New Connection\*\*: Time and resource-intensive for each operation. Suitable for applications with infrequent database access or low traffic.

- \*\*Using a Connection Pool\*\*: Reduces overhead by reusing existing connections. Ideal for applications with frequent database interactions and high traffic.

In essence, while using a connection from a connection pool may seem similar to creating a new connection, the key difference lies in the reusability and pre-established state of pooled connections, which significantly boosts performance and reduces the load on the database server.

**새 연결은 프로그램 실행 시간, 리소스 할당, 네트워크 재설정 등의 오버헤드를 발생시키는데 반해 커넥션 풀은 한 번의 실행으로 위 절차를 모두 끝내고 대신 프로그램이 실행 중인 동안 항상 메모리에 상주하게 된다.**Yes, you are correct. A connection pool is typically maintained in memory while the program is running. The key points about connection pools being loaded in memory are:

1. \*\*Constant Availability\*\*: Connection pools are designed to keep a set of open database connections ready for use at any time. This means that these connections are held in the application's memory space, allowing for quick and efficient access whenever a database interaction is needed.

2. \*\*Memory Management\*\*: **While maintaining connections in memory might seem resource-intensive, it is actually a more efficient use of resources in scenarios where database interactions are frequent.** The memory overhead is offset by the performance gains from reusing connections instead of opening and closing them for each database interaction.

3. \*\*Configurable Size\*\*: Connection pools are typically configurable, allowing you to set the maximum number of connections that can be open and stored in the pool. This helps manage memory usage effectively. The pool size is usually optimized based on the application's needs and the expected database load.

4. \*\*Idle Connections Management\*\*: Most connection pool implementations also manage idle connections to prevent resource wastage. Connections that have not been used for a certain period can be closed automatically, freeing up resources. New connections are created as needed.

5. \*\*Application Lifecycle\*\*: The connection pool is usually initialized when the application starts and is kept alive throughout the application's lifecycle. When the application shuts down, the pool is usually closed, and all the connections are properly closed and released.

6. \*\*Performance Trade-Offs\*\*: The trade-off for having a connection pool in memory is a slight increase in memory usage in exchange for a significant gain in performance, especially under high-load scenarios.

In summary, while a connection pool does occupy memory while the program is running, it is a deliberate design choice that balances memory usage against performance benefits. The ability to quickly serve database requests without the overhead of establishing new connections makes connection pools an essential feature in high-performance and scalable applications.

일반적으로 커넥션 풀은 HikariCP 라이브러리를 사용한다.

**DAO(Data Access Object)**

데이터를 전문적으로 처리하는 객체이다. 일반적으로 데이터베이스의 접근과 처리를 전담하는 객체를 말한다. VO(Value Object)를 단위로 처리한다. DAO를 호출하는 객체는 DAO가 내부에서 어떤 식으로 데이터를 처리하는지 알 수 없게 구성한다. JDBC 프로그램을 작성한다는 의미는 실제로 DAO를 작성한다는 의미가 된다.

**VO 또는 엔티티**

한 튜플은 객체 단위가 된다. 이 튜플을 단위로 하는 객체를 데이터베이스에서는 엔티티라고 한다. 자바에서 이를 처리하기 위해 테이블과 유사한 구조의 클래스를 만들어 튜플을 객체로 처리하는데 이때 만든 객체를 값을 보관하는 객체라고 해서 Value Object라고 한다. VO는 DTO와 비슷하지만 다르다. DTO가 게터와 세터를 모두 제공해 데이터의 변형과 출력을 모두 지원한다면 VO는 일반적으로 게터만 지원한다. DTO는 애플리케이션과 시스템 계층 등 계층 사이를 오가며 데이터를 전달하지만 VO는 데이터 액세스 계층에서만 일한다.

**DTO와 DAO의 차이**

### DTO (Data Transfer Object)

- \*\*Purpose\*\*: A DTO is used to encapsulate data and send it from one part of a system to another. DTOs are typically used to transfer data between layers and services in an application.

- \*\*Structure\*\*: A DTO is a simple, flat object that contains properties but usually does not contain any business logic. It's like a data structure with public fields or getters and setters.

- \*\*Usage\*\*: DTOs are used to transport data, especially when you need to send a batch of data or a view of data from your server to a client (like in a web or mobile application). They help to decouple the presentation layer from the business logic and data access layers.

- \*\*Example\*\*: In a web application, you might use a DTO to send user profile data from the server to the client without exposing the underlying database entities directly.

### DAO (Data Access Object)

- \*\*Purpose\*\*: A DAO is used as an intermediary between the application and the database. It abstracts and encapsulates all access to the data source.

- \*\*Functionality\*\*: DAOs handle all interactions with the data source to obtain and store data. This includes CRUD (Create, Read, Update, Delete) operations.

- \*\*Usage\*\*: DAOs are used within the data access layer of an application. They keep the data access code in a separate layer and isolated from the business logic.

- \*\*Example\*\*: In an application, a `UserDAO` might provide methods like `getUserById`, `saveUser`, `deleteUser`, etc., which internally execute the necessary database operations.

### Key Differences in Usage

- \*\*Layer of Operation\*\*: DTOs operate primarily at the layer where data is transferred between different parts of the system (like between the server and the client, or between different application layers), while DAOs operate at the layer where data is accessed and manipulated (the data access layer).

- \*\*Concerns\*\*: DTOs are concerned with the efficient and safe transportation of data without exposing internal structures or entities. DAOs are concerned with how to efficiently access and manipulate data from a data source.

- \*\*Dependencies\*\*: Typically, DAOs might return or accept DTOs. For example, a `UserDAO` might return a `UserDTO` when retrieving user data.

- \*\*Design Principle\*\*: DTO follows the principle of separating the data representation from the business logic, whereas DAO follows the principle of separating the data persistence logic from the business logic.

Understanding these differences is key in architecting an application with clear separation of concerns, improving maintainability, scalability, and clarity of the codebase.

DAO와 DTO를 사용해 개발해보고 개념들을 복습하자.

**try-with-resources**

자바 7 버전에 추가된 기능으로 트라이 문의 소괄호 안에 사용할 객체들을 모두 집어넣어 사용하는 방식이다. JDBC 커넥션을 불러와 사용하거나 파일 등을 사용할 때 사용할 수 있다. 일반적인 상황 뿐 아니라 예외 발생 시에 자동 close() 기능을 지원한다. 트라이 문이 끝나면 모든 리소스가 자동으로 닫히므로 유용하지만 가독성을 떨어뜨린다. 롬복의 @Cleanup 애노테이션으로 어느 정도 대체 가능하다. 용도에 맞게 잘 구분해 쓰자.

**DTO와 VO**

JDBC와 MVC 패턴, 롬복만을 사용해서 DTO와 VO를 작성할 때는 롬복 애노테이션 외에 DTO와 VO에 큰 차이가 없는 것처럼 보인다. DTO는 @Data 어노테이션을 활용해 게터, 세터, 투스트링, 이퀄스, 해쉬코드 등의 메서드를 더 가지고 VO는 읽기 전용으로 게터만을 가진다는 정도의 차이가 생긴다.

DTO와 VO의 속성이 같은 경우 굳이 두 객체를 따로 만들어야 하는가?하는 문제가 생긴다. 두 객체를 다른 객체로 변환하기가 번거롭고 코드의 양이 증가하기 때문이다. 하지만 나중에 사용하게 될 JPA에서는 필수적으로 DTO와 VO를 구분해야 한다. 또 스프링에서는 DTO의 검증, 변환 등에 전혀 다를 애노테이션들이 필요하기 때문에 둘을 구분해 작성해두자.

일반적으로 번거로운 DTO -> VO, VO -> DTO 변환은 ModelMapper 라이브러리를 이용해 처리한다. 모델매퍼는 게터, 세터를 이용해 객체의 정보를 다른 객체로 복사하는 기능을 제공한다. 빌드 툴의 의존성에 추가해 사용할 수 있다.

**Log4j2**

Log4j는 보안 이슈가 있어 사용하면 안 된다. Log4j2의 핵심 개념은 로그 레벨과 어펜더(Appender)이다. 어펜더는 로그를 어떤 방식으로 기록할 것인지를 의미한다. 어펜더에 따라 콘솔 출력과 파일 출력으로 나뉜다. System.out.println() 대신 콘솔 어펜더를 사용하면 로그가 콘솔에 표현된다. 로그의 레벨은 중요도 개념과 같다. 개발 시에는 많은 로그를 사용할 수 있지만 운영 시에는 개발에 필요한 모든 로그들을 표현할 필요가 없다. 따라서 개발 시에는 로그 레벨을 낮게 해 사용하고 운영 시에는 높은 레벨의 로그들만 표현되도록 설정하는 경우가 많다.



일반적으로 개발 시에는 Info 이하의 레벨을 사용하고 운영할 때는 Error나 Warn 이상을 사용한다. 위 그림에서 Debug 아래에 Trace 레벨이 하나 더 있다.

의존성을 추가하고 resources 폴더의 log4j2.xml 파일을 생성해 레벨과 어펜더를 설정할 수 있다.

**Todo 관련 정리**

DTO, DAO, VO, enum 객체, Service 클래스

DTO – VO에 @Data 애노테이션이 붙어서 데이터 업데이트(게터, 세터), 생성자 생략 등 데이터 변형에 대한 기본적인 기능이 제공되는 데이터 객체

VO – 게터만 제공되는 읽기 전용 데이터 객체로 데이터베이스에 삽입할 때 VO 형태의 순수한 데이터 객체로 변환(매핑)해주어야 한다.

DAO – 데이터 처리(지금까지는 SQL작업)을 위한 다양한 오퍼레이션이 제공되는 클래스. Vo나 특정 매개변수(킷값)을 받아 SQL 작업을 지원한다.

Enum객체 – 스프링 프레임워크 없이 싱글턴 패턴을 사용하기 위해 enum으로 선언하는 듯 하다. 객체를 여러 개 만들 필요 없이 하나의 객체만 생성해 클래스 이름 자체가 객체 이름처럼 사용될 수 있게 한다.

Service 클래스 – DTO를 VO로 변환하기 위한 매핑 메서드를 지원한다.

**PRG(Post-Redirect-Get) 패턴**

양식을 통해 업데이트 대상을 입력받을 후 Get으로 업데이트 양식이 아닌 화면을 보여주는 패턴이다.

**웹 MVC + JDBC 구조의 단점**

DAO와 Service는 하나이지만 컨트롤러는 여러 개이다.

비슷한 컨트롤러를 요구 페이지마다 새로 만들어야 한다. doGet, doPost가 매 컨트롤러마다 비슷한 양상으로 반복된다.

보여주는 JSP만 다르고 동일한 Http 메서드를 처리하는 비슷한 로직들이 많아진다.

예외처리에 대비하기 힘들다.

동일한 리퀘스트 처리나 매핑, 형변환 코드들이 많다.

이런 문제들을 해결하기 위해 객체 지향적 방법론을 적용한 프레임워크들이 만들어졌다. 뒤에서 스프링이 이 문제들을 어떻게 해결하는지 확인하자.

**Stateless로 State의 장점 활용하기**

Http는 비연결성이지만 고객의 상태를 기억하기 특별한 메커니즘을 사용한다. 세션이나 쿠키 토큰 등이 그것이다. 로그인 유지를 위한 모든 기능을 세션 트래킹이라고 한다.

**쿠키**

문자열로 이뤄진 데이터 조각. 요청과 응답 시에 주고받는 형태로 사용. 기본 구조는 키-밸류 구조. 개발자 도구의 애플리케이션 메뉴를 통해 확인할 수 있다.

쿠키의 기본 시나리오

1. 최초 서버 호출 시 해당 서버에서 발행한 쿠키 외에 브라우저는 아무 것도 전송하지 않는다.
2. 서버가 response를 보낼 때 브라우저에게 쿠키를 보내는데 이때 Set-Cookie 라는 HTTP 헤더를 이용한다.
3. 브라우저는 쿠키를 받아 정보를 읽고 이를 파일로 저장하거나 메모리에서만 처리하고 폐기한다. 이때 판단 기준은 쿠키의 유효기간(만료기간)이다.
4. 브라우저가 쿠키를 저장할 경우 브라우저가 서버에 다시 요청을 할 때 HTTP 헤더에 Cookie라는 헤더 이름과 함께 전달한다.(쿠키는 path를 지정할 수 있어 해당 경로에 맞는 쿠키가 전송된다.)
5. 서버에서 필요에 따라 브라우저가 보낸 쿠키를 읽고 활용한다.

**쿠키의 종류**

1. 자동 발행 쿠키: 특별히 정해두지 않은 경우 WAS에 의해 자동 발행된다. 서블릿 컨테이너인 톰캣은 ‘JSESSIONID’라는 이름을 사용.

-서버에서 발행한 쿠키는 브라우저 메모리 상에 보관되어 브라우저 종료시 삭제됨

-서버 발행 쿠키는 경로가 ‘/’로 지정

1. 개발자가 생성하는 쿠키: 서버 발행 쿠키와 다음과 같은 차이가 있다.

* 이름을 원하는대로 지정
* 유효기간 설정 가능(유효기간이 있으면 브라우저가 이를 파일 형태로 보관한다.)
* 반드시 직접 response에 추가해야 함
* 경로나 도메인을 지정할 수 있다.(특정 서버의 경로를 호출하는 경우 쿠키 사용)

**서블릿 컨텍스트와 세션 저장소**

하나의 톰캣은 여러 개의 웹 애플리케이션을 실행할 수 있다. 실제 운영 시 각각의 웹 애플리케이션은 별도의 도메인으로 분리되어 운영된다. 각 애플리케이션은 자신만의 고유한 메모리 영역을 가지는데 여기에서 서블릿이나 JSP를 인스턴스로 만들어 서비스를 제공한다. 이 영역을 서블릿 API의 서블릿 컨텍스트라고 한다. 이때 세션 쿠키를 관리하기 위한 메모리 영역이 하나 더 생성되는데 이 공간을 세션 저장소라고 한다. 세션 저장소는 키-밸류를 보관한다. 키가 되는 것이 톰캣에서는 JSESSIONID이다. 톰캣의 세션 저장소는 발행한 세션 쿠키들의 정보를 보관한다. 따라서 JSESSIONID가 많아질수록 메모리를 많이 점유하게 된다. 과부하를 피하기 위해 톰캣은 주기적으로 세션 저장소를 조사하며 더이상 사용하지 않는 값들을 정리한다. Session-timeout 설정은 지정된 시간보다 오래된 값들을 주기적 검사마다 삭제한다. 톰캣은 30분 설정

**세션을 통한 상태 유지 메커니즘**

HttpServletRequest의 getSession() 메서드를 실행하면 톰캣에서 JSESSIONID의 쿠키가 요청할 때 있었는지 확인하고 없다면 새 값을 만들어 세션 저장소에 보관한다. 각 JSESSIONID는 다시 키마다 고유한 공간을 가지는데 여기에 사용자들마다 다른 키-값, 서블릿/JSP 객체를 보관할 수 있다. 여기에 해당 사용자의 고유키 – 로그인 상태가 기록되어 있으면 사용자의 로그인 여부를 인정하는 용도로 사용할 수 있다. 서블릿 API에서는 HttpServeltRequest의 getSession() 메서드로 각 JSESSIONID에 접근할 수 있다. HttpSession 타입 객체를 이용하면 현재 사용자만의 공간에 원하는 객체를 저장하거나 수정/삭제할 수 있다. inNew()는 새 공간을 만들어 낸 것인지 기존 공간을 재사용하는 것인지 알려준다.

**필터를 이용한 로그인 체크**

로그인 여부를 확인하려 서블릿 컨트롤러마다 동일하게 체크 로직을 작성하면 같은 코드를 반복해야 한다. 이런 수고를 덜기 위해 서블릿 필터라는 것을 이용한다. 서블릿 필터는 서블릿 API의 특별한 객체로 @WebFilter 애노테이션을 이용해 특별한 경로에 접근할 때 필터가 동작하도록 설계한다. 여러 개의 필터를 적용할 수 있어 특정 서블릿에 도달하기 위한 절차를 만들 수도 있다.

**사용자 정의 쿠키**

개발자가 직접 newCookie()로 생성해 반드시 HttpServletResponse에 addCookie()로 적용해야 전송 가능하다. 초 단위로 유효기간을 정할 수 있고 유효기간이 없으면 브라우저 메모리에만 보관, 있으면 브라우저가 파일, 기타 방식으로 저장한다. 세션 쿠키와 동일하게 4kb이다.

쿠키는 서버와 브라우저를 오가기 때문에 보안에 취약하다. ‘오늘 하루 이 창 열지 않기’나 ‘최근 본 상품 목록’ 등 **사소하고 서버에서 보관할 필요가 없는 데이터**는 쿠키를 이용해 처리하자. 모바일에서 자동 로그인으로 쿠키의 활용도가 늘었다는 점을 알아두자.

**쿠키를 이용한 자동 로그인(remember-me)**

로그인 사용자의 정보를 쿠키에 보관하고 이를 이용해 사용자 정보를 HttpSession에 담는 방식이다. 원래 많은 것을 고려해야 하지만 뒤에 스프링 부트와 시큐리티에서 다루고 여기서는 간단히 아이디어를 검증한다.

간단한 로그인 아이디어

* 사용자가 로그인할 때 임의의 문자열을 생성해 데이터베이스에 보관한다.
* 쿠키에 생성된 문자열을 값으로 삼고 유효기간을 1주일 지정한다.

로그인 체크는 다음 방식으로 구현

* 현재 사용자의 HttpSession에 로그인 정보가 없는 경우에만 쿠키를 확인
* 쿠키의 값과 데이터베이스 값을 비교하고 같으면 사용자 정보를 읽어와 HttpSession에 사용자 정보를 추가한다.

위 아이디어에서 쿠키값을 탈취당하면 문제가 발생하기 때문에 주기적으로 쿠키의 값을 갱신하는 부분이 추가되어야 한다. UUID(Universally Unique Identifier)를 이용한 임의의 문자열을 이용하기로 한다. UUID는 범용 고유 식별자라고도 하며 고유한 번호를 랜덤으로 생성할 때 사용한다. 자바는 java.util패키지를 이용해 처리 가능하다.

**지금까지 작업 흐름 정리**

**VO(순수 데이터) -> DAO[커넥션풀 + DAO클래스로 구성](DB의 속성값이나 쿼리 조건절에 넣을 인자를 받아 쿼리를 만들어낸다. 이후 커넥션 풀을 활용해 DB에 쿼리를 전송하고 resultSet와 빌더 패턴을 활용해 DB에 반영된 사항으로 VO 객체를 생성한다.) -> Service(DAO객체와 매퍼를 멤버변수로 가진다. 비즈니스 로직을 실행하며 이를 위해 DTO->VO, VO->DTO 전환 등을 DAO객체와 매퍼를 통해 수행한다.) -> Controller(JSP에서 수행되는 고객 요구 사항에 반응하기 위해 http 프로토콜과 service를 활용한다.)**

**서블릿 리스너**

서블릿 API가 지원하는 특별한 인터페이스이다. 이벤트라는 특정 데이터가 발생하면 자동으로 리스너 객체가 실행된다. 이벤트가 발생할 때 약속해둔 동작을 수행한다는 점을 이용해 기존 코드를 변경하지 않고도 추가적인 기능을 수행할 수 있게 한다. 스프링MVC가 리스너를 통해 동작한다.

* 애플리케이션 시작과 종료에 특정 작업 수행
* HttpSession에 특정 작업 감시, 처리
* HttpServletRequest에 특정 작업 감시, 처리

등은 서블릿 API가 정의하는 리스너 인터페이스의 정의에 부합하는 작업들이다.

서블릿 리스너를 이용해 커넥션 풀을 초기화하거나 ServletContext를 이용해 프로젝트가 실행될 때 필요한 객체들을 준비하는 작업 등을 할 수 있다. 스프링 프레임워크를 웹 프로젝트에서 미리 로딩해둘 때 ServletContextListener를 이용한다.

**-----------------------------------------서블릿+JSP+JDBC(스프링 이전)--------------------------------**

**스프링**

스프링은 의존성 주입을 구현하는 객체지향 프레임워크로 출발했다. 스프링 프레임워크는 핵심 역할을 하는 코어 라이브러리와 여러 개의 부수적인 라이브러리를 결합하는 형태로 프로젝트를 구성한다. 대표적인 Spring Web MVC나 JDBC 처리용 MyBatis를 연동하는 mybatis-spring과 같은 라이브러리가 그러한 예이다.

**의존성 주입**

어떻게 하면 객체와 객체의 관계를 더 유연하게 유지할 것인가?하는 고민에서 출발한다. 스프링 이전 프로젝트에서 모든 컨트롤러는 비즈니스 로직을 사용하기 위해 서비스 객체들을 이용해야만 했는데 이 경우 컨트롤러가 서비스에 의존적이었다. 의존성을 해결하기 위해 컨트롤러에 서비스 객체를 직접 생성하거나 싱글턴 패턴을 활용하는 등 다양한 패턴을 설계해 적용할 수 있다. 스프링은 이런 의존성 문제를 프레임워크 자체에서 지원한다. 이때 다양한 방식으로 객체를 찾아서 사용하도록 XML이나 자바 설정을 이용할 수 있다.

**빈**

스프링 빈은 선 마이크로 시스템즈가 개발한 자바빈즈 컴포넌트 아키텍처에서 왔다. 빈은 어감대로 작고, 간단하며 관리가 쉬운 작은 단위의 컴포넌트를 의미한다. 스프링에서는 스프링 컨테이너가 관리하는 싱글턴 객체가 된다. 이를 위해 XML이나 별도의 클래스를 이용해 빈 관련 설정을 해준다.

인텔리제이에서는 WEB-INF이하에 root-context.xml을 생성하면 이를 스프링 빈 설정 xml로 인식해 Configure application context라는 설정 메뉴가 생긴다. 해당 xml 파일을 Root Context로 지정하고 xml에 빈 설정 스크립트를 넣어주면 원하는 클래스를 빈 객체로 다룰 수 있다.

스프링 프로젝트는 상당히 많은 객체를 생성하기 떄문에 한 번 에러가 발생하면 원인을 찾기 상당히 힘든 경우가 많다. 가능하면 개발 단계에서 많은 테스트를 하며 개발하는 것이 좋다.

**애플리케이션 컨텍스트**

이전 장에서 서블릿 컨테이너가 관리하는 서블릿 컨텍스트에 대해 배웠었다. 스프링에서는 빈을 관리하기 위한 ApplicationContext가 존재한다. root-context.xml을 만들었다면 스프링 실행 후 해당 xml을 이용해 ApplicationContext 객체가 생성된다. 위 xml 파일에서 <bean>으로 지정된 객체는 ApplicationContext 안에 생성되어 관리되기 시작한다.

런타임 시에는 클래스를 작성하거나 객체를 직접 생성하지 않는다. 이 역할은 스프링 컨테이너의 ApplicationContext에 의해 이뤄져야 한다. 과거에는 서블릿도 xml 파일에 <servlet> 태그를 이용해 서블릿으로 지정할 클래스명과 경로를 모두 지정해줘야 했지만 이를 @WebServlet이 대체하는 방식으로 발전했다. 스프링도 비슷하다. 2.5버전 이후에는 xml 지정 방식이 애노테이션 방식으로 변환되며 편리한 설정이 가능해졌다.

* @Controller
* @Service
* @Repository: DAO와 같은 객체용 어노테이션
* @Component: 일반 객체나 유틸리티 객체를 위한 어노테이션

스프링이 원래 웹만을 위한 프레임워크가 아닌 것처럼 어노테이션 프로세싱은 웹 영역 뿐만 아니라 애플리케이션이 미치는 다양한 영역의 객체에 적용할 수 있다.

@Autowired를 멤버변수에 할당하는 방식은 필드 주입 방식이다.

**생성자 주입 방식**

스프링 3 이후에는 필드 주입보다 생성자 주입 방식이 더 선호되었다. 생성자 주입은 다음과 같은 규칙으로 작성한다.

* 주입 받아야 하는 변수는 final로 작성한다.
* 생성자를 이용해 해당 변수를 생성자의 파라미터로 지정한다.

이 방식이 필드 주입이나 Setter 주입보다 선호되는 이유는 객체를 생성할 때 문제가 발생할지 미리 확인해볼 수 있기 때문이다. Lombok에서는 @RequiredArgsConstructor을 이용해 필요한 생성자 함수를 자동으로 작성할 수 있다.

**구체 클래스에 대한 의존을 인터페이스에 의존하도록 만들어 결합도 낮추기**

위에서 생성자를 통한 의존성 주입이 필드 주입보다 선호되어지는 이유에 대해 알아봤다. 하지만 생성자 주입 방식에도 여전히 한계가 있다. 생성자를 주입 받은 객체가 구체적인 클래스를 타입으로 갖게 되면 나중에 타입으로 지정된 클래스를 변경하게 될 때 집합연관관계에 있는 클래스도 변경해줘야 한다. 이 문제는 추상화된 타입을 통해 해결할 수 있다. 인터페이스를 객체 타입으로 지정해 생성자 주입을 하게 되면 구체 클래스를 캡슐화 해 드러내지 않고도 실제 객체를 구현할 수 있게 된다.

인터페이스를 구현한 클래스를 스프링 빈으로 등록하고 인터페이스 타입으로 객체를 생성해 사용하면 실제 객체를 모르고도 코드를 작성할 수 있다.(구체가 아닌 추상에 의존 -> 느슨한 결합(loose coupling)). 느슨한 결합을 이용하면 참조 당하는 객체의 코드를 수정할 때 참조하는 객체를 사용하는 다른 클래스의 코드를 수정할 필요가 없게 된다.

주의해야할 점 -> 의존되는 인터페이스를 구현하는 객체가 여러 개이면 충돌이 발생해 에러가 난다. 애플리케이션 컨텍스트는 해당 인터페이스를 구현하는 빈이 하나인 경우에만 제대로 동작한다. 따라서 하나의 인터페이스를 구현하는 두 개 이상의 빈을 두고 쓰는 경우에는 우선하는 빈에 @Primary 어노테이션을 붙여 사용해야 한다. @Qualifier를 이용하는 방식도 있다. 퀄러파이어는 이름을 지정해 특정 이름의 객체를 주입받는 방식이다. 롬복과 같이 쓰기 위해 lombok.config을 메인 소스 패키지의 자바 폴더에 생성하고 다음과 같이 작성한다.

lombok.copyableAnnotations += org.springframework.beans.factory.annotation.Qualifier

스프링 빈으로 등록되는 개체들은 MVC 내에서 핵심 배역을 맡아 프로그램 메모리에 오래 상주하는 ‘역할’ 중심 객체들이다. 역할보다 데이터가 중심이 되는 VO나 DTO 등은 빈에 해당하지 않는다. 생각해보면 생명주기가 짧은 DTO는 메모리에 오랫동안 둘 필요가 없다.

생성자 주입을 통해 빈을 만드는 이점에 대해 살펴봤지만 그럼에도 XML이나 필드 주입을 활용해 의존성 주입을 구현하는 경우가 있다. 이에 대한 기준은 ‘코드를 수정할 수 있는가’가 된다. jar파일로 추가되는 클래스의 객체를 스프링의 빈으로 처리해야 한다면 해당 코드가 존재하지 않기 때문에 어노테이션을 추가할 수 없는 문제가 생긴다. 이런 객체들은 XML의 <bean>을 이용해 처리하고 직접 장성되는 클래스는 어노테이션을 이용하는 게 좋다.