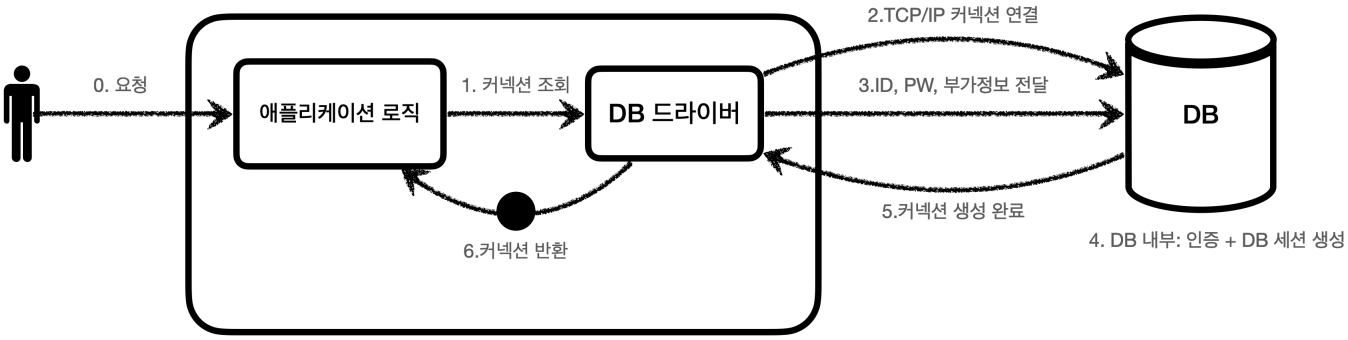
# 2. 커넥션풀과 데이터소스 이해

#1.인강/7.스프링 DB 1/강의#



## 커넥션 풀 이해

**데이터베이스 커넥션을 매번 획득**




데이터베이스 커넥션을 획득할 때는 다음과 같은 복잡한 과정을 거친다.

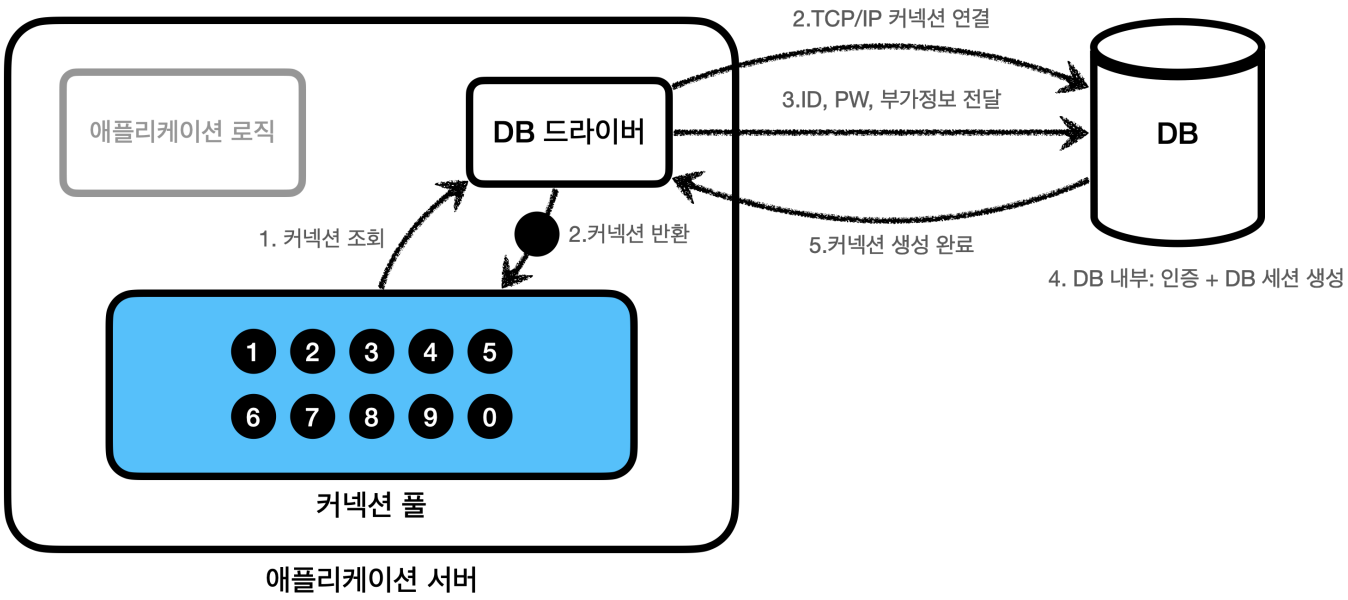
1. 애플리케이션 로직은 DB 드라이버를 통해 커넥션을 조회한다.
2. DB 드라이버는 DB와 `TCP/IP` 커넥션을 연결한다. 물론 이 과정에서 3 way handshake 같은 `TCP/IP` 연결을 위한 네트워크 동작이 발생한다.
3. DB 드라이버는 `TCP/IP` 커넥션이 연결되면 ID, PW와 기타 부가정보를 DB에 전달한다.
4. DB는 ID, PW를 통해 내부 인증을 완료하고, 내부에 DB 세션을 생성한다.
5. DB는 커넥션 생성이 완료되었다는 응답을 보낸다.
6. DB 드라이버는 커넥션 객체를 생성해서 클라이언트에 반환한다.

이렇게 커넥션을 새로 만드는 것은 과정도 복잡하고 시간도 많이 많이 소모되는 일이다.
DB는 물론이고 애플리케이션 서버에서도 `TCP/IP` 커넥션을 새로 생성하기 위한 리소스를 매번 사용해야 한다.
진짜 문제는 고객이 애플리케이션을 사용할 때, SQL을 실행하는 시간 뿐만 아니라 커넥션을 새로 만드는 시간이 추가되기 때문에 결과적으로 응답 속도에 영향을 준다. 이것은 사용자에게 좋지 않은 경험을 줄 수 있다.

> **참고**: 데이터베이스마다 커넥션을 생성하는 시간은 다르다. 시스템 상황마다 다르지만 MySQL 계열은 수 ms(밀리초) 정도로 매우 빨리 커넥션을 확보할 수 있다. 반면에 수십 밀리초 이상 걸리는 데이터베이스들도 있다.

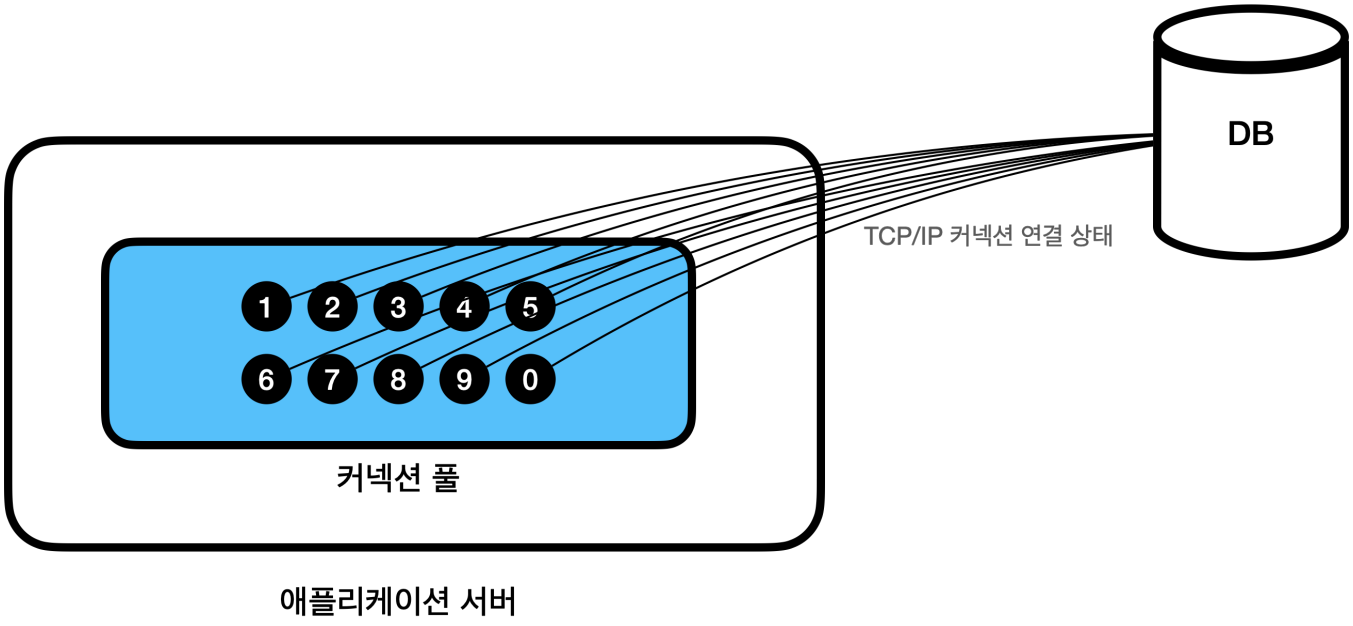
이런 문제를 한번에 해결하는 아이디어가 바로 커넥션을 미리 생성해두고 사용하는 커넥션 풀이라는 방법이다.
커넥션 풀은 이름 그대로 커넥션을 관리하는 풀(수영장 풀을 상상하면 된다.)이다.

**커넥션 풀 초기화**




애플리케이션을 시작하는 시점에 커넥션 풀은 필요한 만큼 커넥션을 미리 확보해서 풀에 보관한다. 보통 얼마나 보관할지는 서비스의 특징과 서버 스펙에 따라 다르지만 기본값은 보통 10개이다.

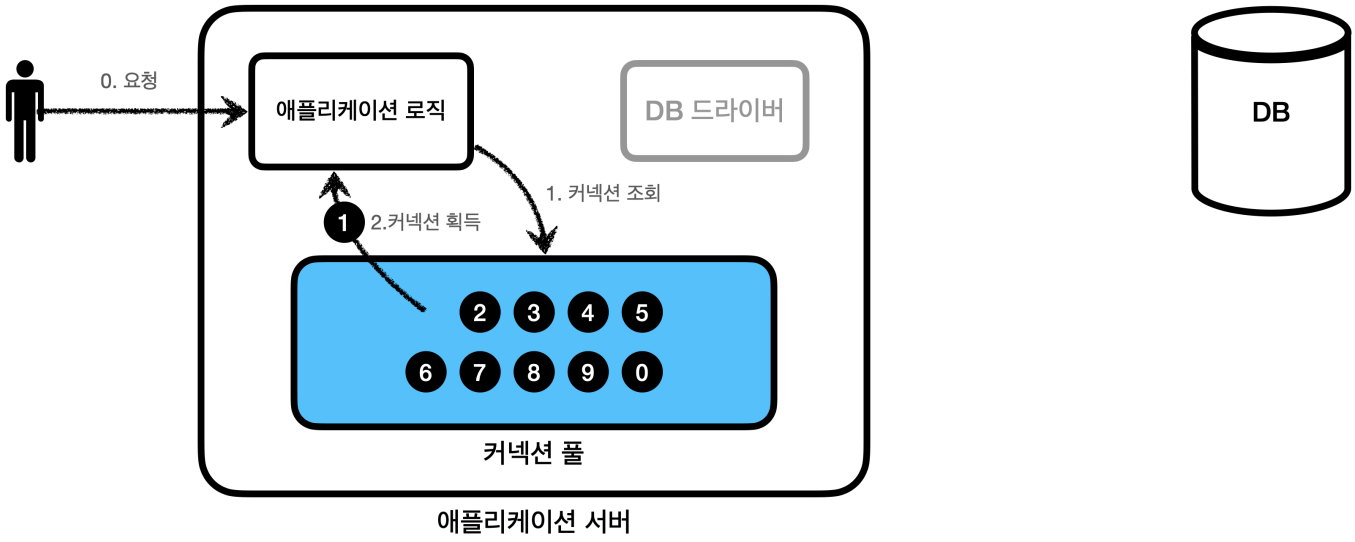
**커넥션 풀의 연결 상태**




커넥션 풀에 들어 있는 커넥션은 TCP/IP로 DB와 커넥션이 연결되어 있는 상태이기 때문에 언제든지 즉시 SQL을 DB

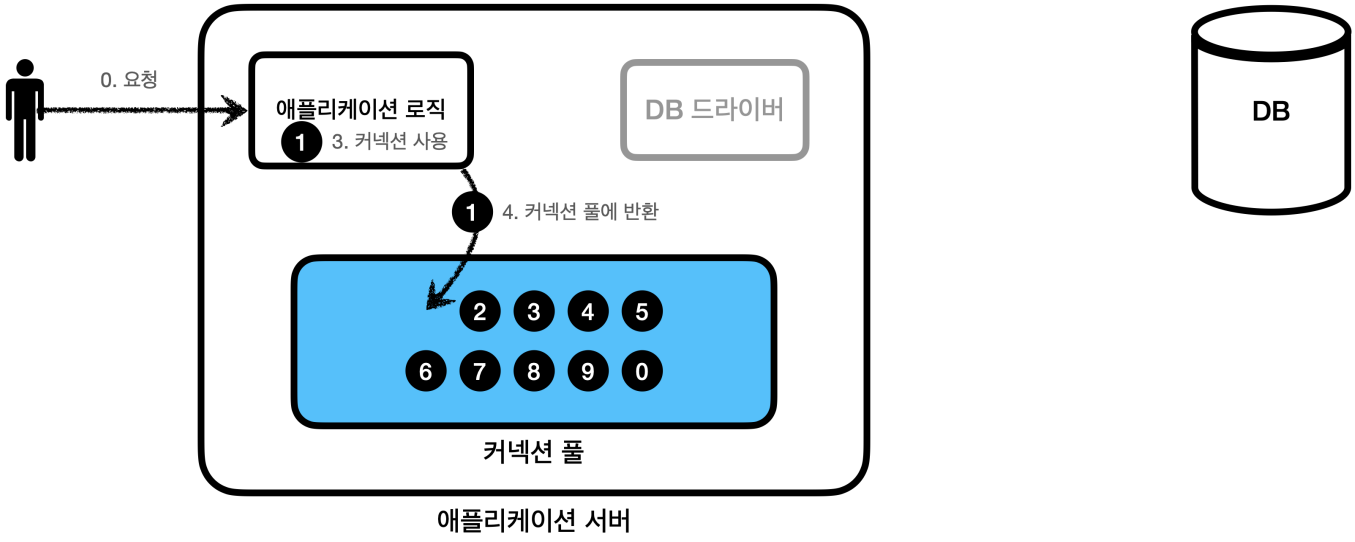
에 전달할 수 있다.

**커넥션 풀 사용1**




- 애플리케이션 로직에서 이제는 DB 드라이버를 통해서 새로운 커넥션을 획득하는 것이 아니다.
- 이제는 커넥션 풀을 통해 이미 생성되어 있는 커넥션을 객체 참조로 그냥 가져다 쓰기만 하면 된다.
- 커넥션 풀에 커넥션을 요청하면 커넥션 풀은 자신이 가지고 있는 커넥션 중에 하나를 반환한다.

**커넥션 풀 사용2**




- 애플리케이션 로직은 커넥션 풀에서 받은 커넥션을 사용해서 SQL을 데이터베이스에 전달하고 그 결과를 받아서 처리한다.
- 커넥션을 모두 사용하고 나면 이제는 커넥션을 종료하는 것이 아니라, 다음에 다시 사용할 수 있도록 해당 커넥션을 그대로 커넥션 풀에 반환하면 된다. 여기서 주의할 점은 커넥션을 종료하는 것이 아니라 커넥션이 살아있는 상태로 커넥션 풀에 반환해야 한다는 것이다.

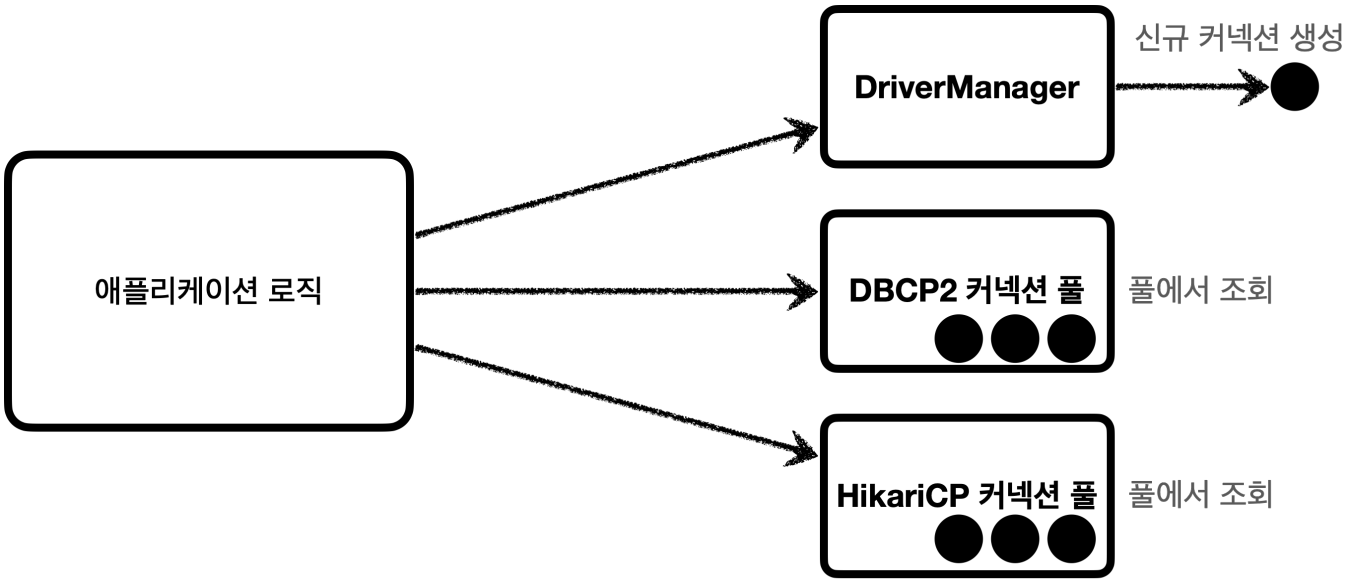
**정리**

- 적절한 커넥션 풀 숫자는 서비스의 특징과 애플리케이션 서버 스펙, DB 서버 스펙에 따라 다르기 때문에 성능 테스트를 통해서 정해야 한다.
- 커넥션 풀은 서버당 최대 커넥션 수를 제한할 수 있다. 따라서 DB에 무한정 연결이 생성되는 것을 막아주어서 DB를 보호하는 효과도 있다.
- 이런 커넥션 풀은 얻는 이점이 매우 크기 때문에 **실무에서는 항상 기본으로 사용**한다.
- 커넥션 풀은 개념적으로 단순해서 직접 구현할 수도 있지만, 사용도 편리하고 성능도 뛰어난 오픈소스 커넥션 풀이 많기 때문에 오픈소스를 사용하는 것이 좋다.
- 대표적인 커넥션 풀 오픈소스는 `commons-dbcp2`, `tomcat-jdbc pool`, `HikariCP` 등이 있다.
- 성능과 사용의 편리함 측면에서 최근에는 `hikariCP`를 주로 사용한다. 스프링 부트 2.0 부터는 기본 커넥션 풀로 `hikariCP`를 제공한다. 성능, 사용의 편리함, 안전성 측면에서 이미 검증이 되었기 때문에 커넥션 풀을 사용할 때는 고민할 것 없이 `hikariCP`를 사용하면 된다. 실무에서도 레거시 프로젝트가 아닌 이상 대부분 `hikariCP`를 사용한다.

## DataSource 이해

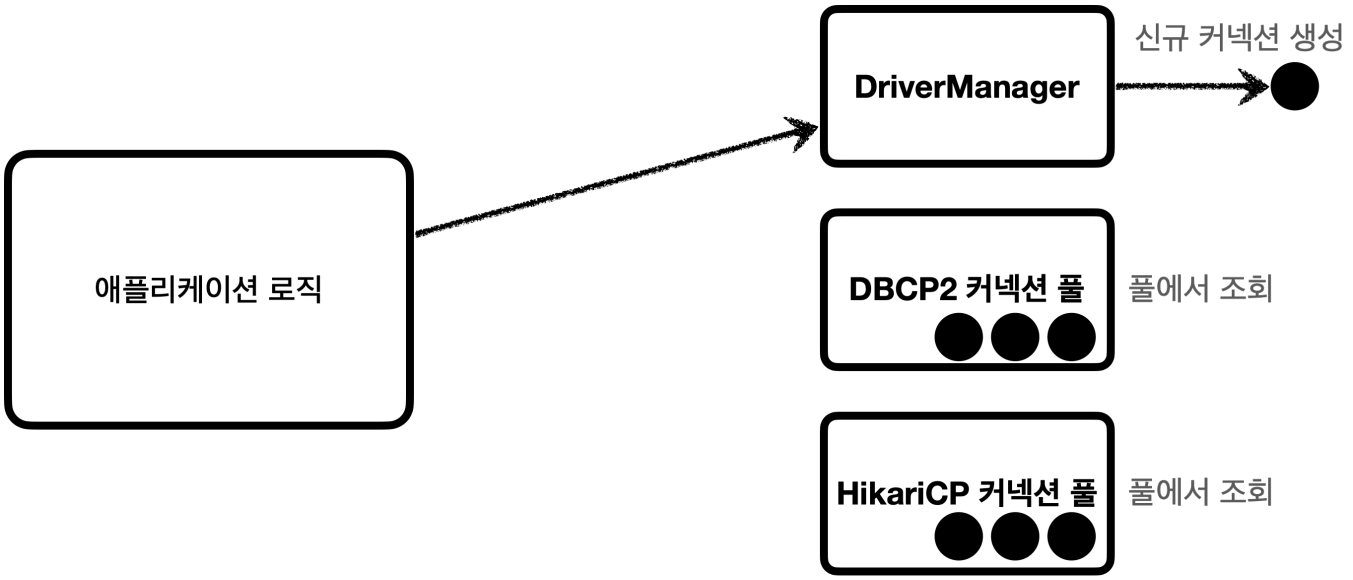
커넥션을 얻는 방법은 앞서 학습한 JDBC `DriverManager`를 직접 사용하거나, 커넥션 풀을 사용하는 등 다양한 방법이 존재한다.

**커넥션을 획득하는 다양한 방법**




커넥션을 획득하는 다양한 방법

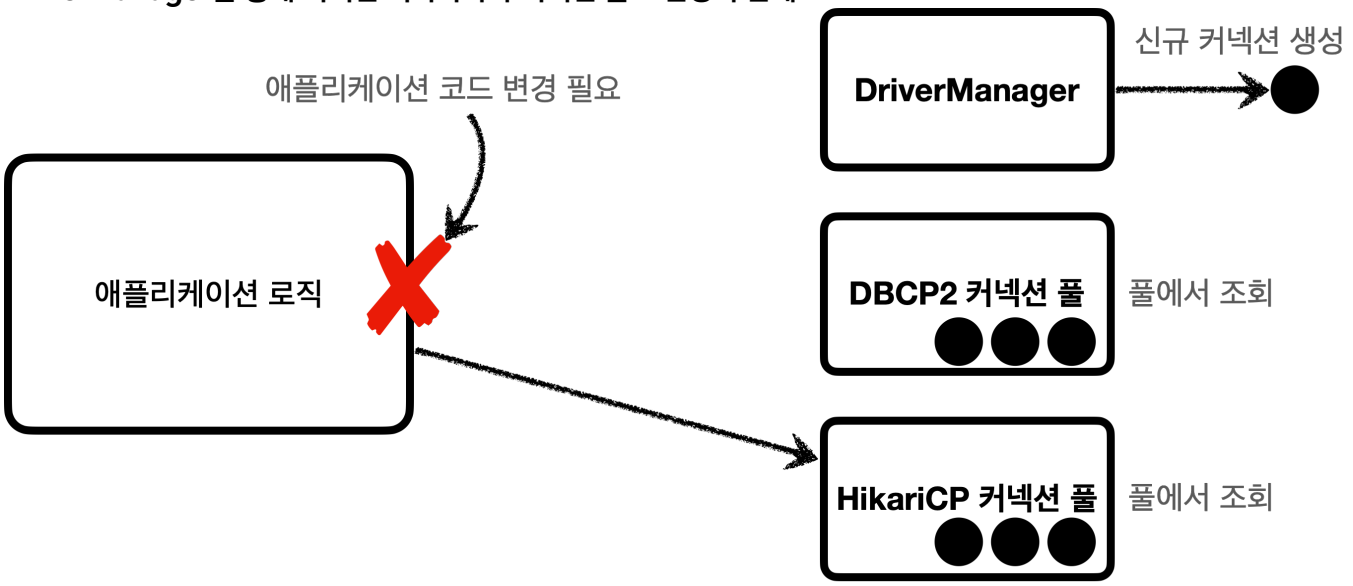
**DriverManager를 통해 커넥션 획득**




커넥션을 획득하는 방법의 변경

- 우리가 앞서 JDBC로 개발한 애플리케이션 처럼 `DriverManager`를 통해서 커넥션을 획득하다가, 커넥션 풀을 사용하는 방법으로 변경하려면 어떻게 해야할까?

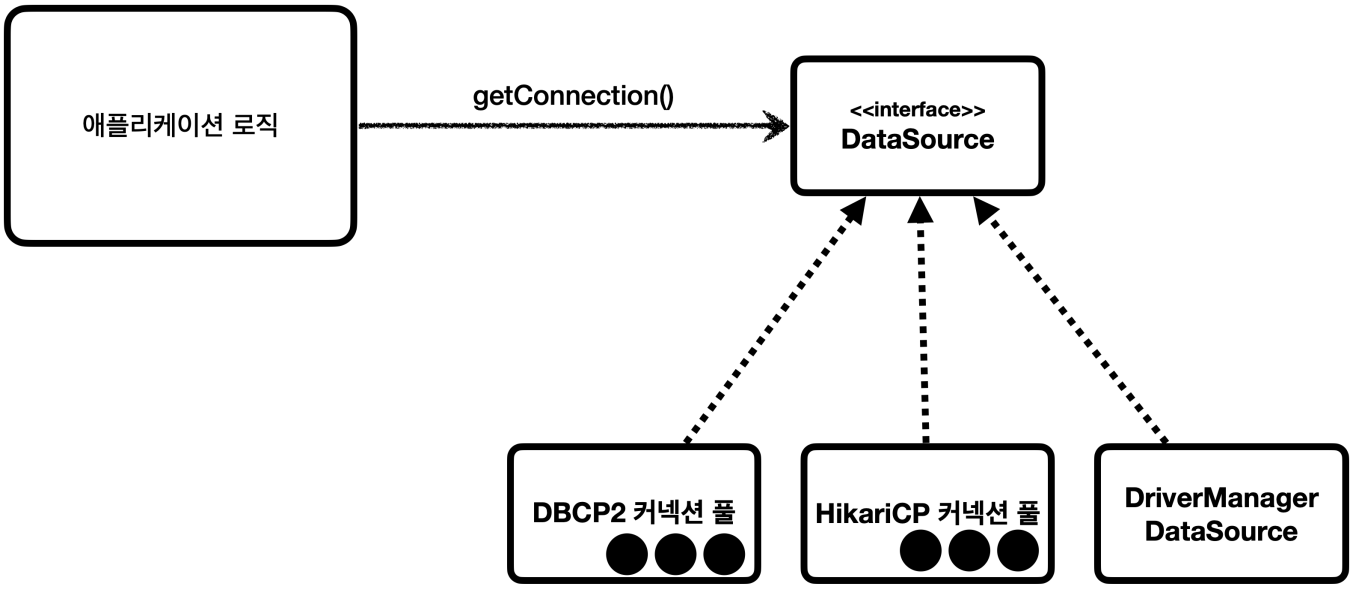
**DriverManager를 통해 커넥션 획득하다가 커넥션 풀로 변경시 문제**




커넥션을 획득하는 방법의 변경

- 예를 들어서 애플리케이션 로직에서 `DriverManager`를 사용해서 커넥션을 획득하다가 `HikariCP` 같은 커넥션 풀을 사용하도록 변경하면 커넥션을 획득하는 애플리케이션 코드도 함께 변경해야 한다. 의존관계가 `DriverManager`에서 `HikariCP`로 변경되기 때문이다. 물론 둘의 사용법도 조금씩 다를 것이다.

**커넥션을 획득하는 방법을 추상화**




커넥션을 획득하는 방법을 추상화

- 자바에서는 이런 문제를 해결하기 위해 `javax.sql.DataSource`라는 인터페이스를 제공한다.
- `DataSource`는 **커넥션을 획득하는 방법을 추상화** 하는 인터페이스이다.
- 이 인터페이스의 핵심 기능은 커넥션 조회 하나이다. (다른 일부 기능도 있지만 크게 중요하지 않다.)

### DataSource 핵심 기능만 축약

```
public interface DataSource {
  Connection getConnection() throws SQLException;
}
```

### 정리

- 대부분의 커넥션 풀은 `DataSource` 인터페이스를 이미 구현해두었다. 따라서 개발자는 `DBCP2 커넥션 풀`, `HikariCP 커넥션 풀`의 코드를 직접 의존하는 것이 아니라 `DataSource` 인터페이스에만 의존하도록 애플리케이션 로직을 작성하면 된다.
- 커넥션 풀 구현 기술을 변경하고 싶으면 해당 구현체로 갈아끼우기만 하면 된다.
- `DriverManager`는 `DataSource` 인터페이스를 사용하지 않는다. 따라서 `DriverManager`는 직접 사용해야 한다. 따라서 `DriverManager`를 사용하다가 `DataSource` 기반의 커넥션 풀을 사용하도록 변경하면 관련 코드를 다 고쳐야 한다. 이런 문제를 해결하기 위해 스프링은 `DriverManager`도 `DataSource`를 통해서 사용할 수 있도록 `DriverManagerDataSource`라는 `DataSource`를 구현한 클래스를 제공한다.
- 자바는 `DataSource`를 통해 커넥션을 획득하는 방법을 추상화했다. 이제 애플리케이션 로직은 `DataSource` 인터페이스에만 의존하면 된다. 덕분에 `DriverManagerDataSource`를 통해서 `DriverManager`를 사용하다가 커넥션 풀을 사용하도록 코드를 변경해도 애플리케이션 로직은 변경하지 않아도 된다.

# DataSource 예제1 - DriverManager

예제를 통해 `DataSource`를 알아보자.

먼저 기존에 개발했던 `DriverManager`를 통해서 커넥션을 획득하는 방법을 확인해보자.

**ConnectionTest - 드라이버 매니저**

```
package hello.jdbc.connection;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import static hello.jdbc.connection.ConnectionConst.*;

@Slf4j
public class ConnectionTest {

    @Test
    void driverManager() throws SQLException {
        Connection con1 = DriverManager.getConnection(URL, USERNAME, PASSWORD);
        Connection con2 = DriverManager.getConnection(URL, USERNAME, PASSWORD);
        log.info("connection={}, class={}", con1, con1.getClass());
        log.info("connection={}, class={}", con2, con2.getClass());
    }
}
```

**실행 결과**

```
connection=conn0: url=jdbc:h2:tcp://..test user=SA, class=class
org.h2.jdbc.JdbcConnection
connection=conn1: url=jdbc:h2:tcp://..test user=SA, class=class
org.h2.jdbc.JdbcConnection
```

이번에는 스프링이 제공하는 `DataSource`가 적용된 `DriverManager`인 `DriverManagerDataSource`를 사용해보자.

**ConnectionTest - 데이터소스 드라이버 매니저 추가**

```
package hello.jdbc.connection;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import static hello.jdbc.connection.ConnectionConst.*;

@Slf4j
public class ConnectionTest {

    @Test
    void driverManager() throws SQLException {
        Connection con1 = DriverManager.getConnection(URL, USERNAME, PASSWORD);
        Connection con2 = DriverManager.getConnection(URL, USERNAME, PASSWORD);
        log.info("connection={}, class={}", con1, con1.getClass());
        log.info("connection={}, class={}", con2, con2.getClass());
    }

    @Test
    void dataSourceDriverManager() throws SQLException {
        //DriverManagerDataSource - 항상 새로운 커넥션 획득
        DriverManagerDataSource dataSource = new DriverManagerDataSource(URL,
USERNAME, PASSWORD);
        useDataSource(dataSource);
    }

    private void useDataSource(DataSource dataSource) throws SQLException {
        Connection con1 = dataSource.getConnection();
        Connection con2 = dataSource.getConnection();
        log.info("connection={}, class={}", con1, con1.getClass());
        log.info("connection={}, class={}", con2, con2.getClass());
    }
}
```

**dataSourceDriverManager() - 실행 결과**

```
DriverManagerDataSource - Creating new JDBC DriverManager Connection to
[jdbc:h2:tcp:..test]
DriverManagerDataSource - Creating new JDBC DriverManager Connection to
[jdbc:h2:tcp:..test]
connection=conn0: url=jdbc:h2:tcp://..test user=SA, class=class
org.h2.jdbc.JdbcConnection
connection=conn1: url=jdbc:h2:tcp://..test user=SA, class=class
org.h2.jdbc.JdbcConnection
```

기존 코드와 비슷하지만 `DriverManagerDataSource`는 `DataSource`를 통해서 커넥션을 획득할 수 있다. 참고로 `DriverManagerDataSource`는 스프링이 제공하는 코드이다.

**파라미터 차이**

기존 `DriverManager`를 통해서 커넥션을 획득하는 방법과 `DataSource`를 통해서 커넥션을 획득하는 방법에는 큰 차이가 있다.

**DriverManager**

```
DriverManager.getConnection(URL, USERNAME, PASSWORD)
DriverManager.getConnection(URL, USERNAME, PASSWORD)
```

**DataSource**

```
void dataSourceDriverManager() throws SQLException {
    DriverManagerDataSource dataSource = new DriverManagerDataSource(URL,
USERNAME, PASSWORD);
    useDataSource(dataSource);
}

private void useDataSource(DataSource dataSource) throws SQLException {
    Connection con1 = dataSource.getConnection();
    Connection con2 = dataSource.getConnection();
    log.info("connection={}, class={}", con1, con1.getClass());
    log.info("connection={}, class={}", con2, con2.getClass());
}
```

- `DriverManager`는 커넥션을 획득할 때 마다 `URL`, `USERNAME`, `PASSWORD` 같은 파라미터를 계속 전달해야 한다. 반면에 `DataSource`를 사용하는 방식은 처음 객체를 생성할 때만 필요한 파리미터를 넘겨두고, 커넥션을 획득할 때는 단순히 `dataSource.getConnection()`만 호출하면 된다.

**설정과 사용의 분리**

- **설정**: `DataSource`를 만들고 필요한 속성들을 사용해서 `URL`, `USERNAME`, `PASSWORD` 같은 부분을 입력하는 것을 말한다. 이렇게 설정과 관련된 속성들은 한 곳에 있는 것이 향후 변경에 더 유연하게 대처할 수 있다.
- **사용**: 설정은 신경쓰지 않고, `DataSource`의 `getConnection()`만 호출해서 사용하면 된다.

**설정과 사용의 분리 설명**

- 이 부분이 작아보이지만 큰 차이를 만들어내는데, 필요한 데이터를 `DataSource`가 만들어지는 시점에 미리 다 넣어두게 되면, `DataSource`를 사용하는 곳에서는 `dataSource.getConnection()`만 호출하면 되므로, `URL`, `USERNAME`, `PASSWORD` 같은 속성들에 의존하지 않아도 된다. 그냥 `DataSource`만 주입받아서 `getConnection()`만 호출하면 된다.
- 쉽게 이야기해서 리포지토리(Repository)는 `DataSource`만 의존하고, 이런 속성을 몰라도 된다.
- 애플리케이션을 개발해보면 보통 설정은 한 곳에서 하지만, 사용은 수 많은 곳에서 하게 된다.
- 덕분에 객체를 설정하는 부분과, 사용하는 부분을 좀 더 명확하게 분리할 수 있다.

# DataSource 예제2 - 커넥션 풀

이번에는 `DataSource`를 통해 커넥션 풀을 사용하는 예제를 알아보자.

**ConnectionTest - 데이터소스 커넥션 풀 추가**

```
import com.zaxxer.hikari.HikariDataSource;

@Test
void dataSourceConnectionPool() throws SQLException, InterruptedException {
    //커넥션 풀링: HikariProxyConnection(Proxy) -> JdbcConnection(Target)
    HikariDataSource dataSource = new HikariDataSource();
    dataSource.setJdbcUrl(URL);
    dataSource.setUsername(USERNAME);
    dataSource.setPassword(PASSWORD);
    dataSource.setMaximumPoolSize(10);
    dataSource.setPoolName("MyPool");

    useDataSource(dataSource);
    Thread.sleep(1000); //커넥션 풀에서 커넥션 생성 시간 대기
}
```

- HikariCP 커넥션 풀을 사용한다. `HikariDataSource`는 `DataSource` 인터페이스를 구현하고 있다.
- 커넥션 풀 최대 사이즈를 10으로 지정하고, 풀의 이름을 `MyPool`이라고 지정했다.

- 커넥션 풀에서 커넥션을 생성하는 작업은 애플리케이션 실행 속도에 영향을 주지 않기 위해 별도의 쓰레드에서 작동한다. 별도의 쓰레드에서 동작하기 때문에 테스트가 먼저 종료되어 버린다. 예제처럼 `Thread.sleep`을 통해 대기 시간을 주어야 쓰레드 풀에 커넥션이 생성되는 로그를 확인할 수 있다.

**실행 결과**

(로그 순서는 이해하기 쉽게 약간 수정했다.)

```
#커넥션 풀 초기화 정보 출력
HikariConfig - MyPool - configuration:
HikariConfig - maximumPoolSize.............................10
HikariConfig - poolName.............................."MyPool"

#커넥션 풀 전용 쓰레드가 커넥션 풀에 커넥션을 10개 채움
[MyPool connection adder] MyPool - Added connection conn0: url=jdbc:h2:..
user=SA
[MyPool connection adder] MyPool - Added connection conn1: url=jdbc:h2:..
user=SA
[MyPool connection adder] MyPool - Added connection conn2: url=jdbc:h2:..
user=SA
[MyPool connection adder] MyPool - Added connection conn3: url=jdbc:h2:..
user=SA
[MyPool connection adder] MyPool - Added connection conn4: url=jdbc:h2:..
user=SA
...
[MyPool connection adder] MyPool - Added connection conn9: url=jdbc:h2:..
user=SA

#커넥션 풀에서 커넥션 획득1
ConnectionTest - connection=HikariProxyConnection@446445803 wrapping conn0:
url=jdbc:h2:tcp://localhost/~/test user=SA, class=class
com.zaxxer.hikari.pool.HikariProxyConnection
#커넥션 풀에서 커넥션 획득2
ConnectionTest - connection=HikariProxyConnection@832292933 wrapping conn1:
url=jdbc:h2:tcp://localhost/~/test user=SA, class=class
com.zaxxer.hikari.pool.HikariProxyConnection

MyPool - After adding stats (total=10, active=2, idle=8, waiting=0)
```

실행 결과를 분석해보자.

**HikariConfig**

`HikariCP` 관련 설정을 확인할 수 있다. 풀의 이름(`MyPool`)과 최대 풀 수(`10`)을 확인할 수 있다.

**MyPool connection adder**

별도의 쓰레드 사용해서 커넥션 풀에 커넥션을 채우고 있는 것을 확인할 수 있다. 이 쓰레드는 커넥션 풀에 커넥션을 최대 풀 수(`10`)까지 채운다.

그렇다면 왜 별도의 쓰레드를 사용해서 커넥션 풀에 커넥션을 채우는 것일까?

커넥션 풀에 커넥션을 채우는 것은 상대적으로 오래 걸리는 일이다. 애플리케이션을 실행할 때 커넥션 풀을 채울 때 까지 마냥 대기하고 있다면 애플리케이션 실행 시간이 늦어진다. 따라서 이렇게 별도의 쓰레드를 사용해서 커넥션 풀을 채워야 애플리케이션 실행 시간에 영향을 주지 않는다.

**커넥션 풀에서 커넥션 획득**

커넥션 풀에서 커넥션을 획득하고 그 결과를 출력했다. 여기서는 커넥션 풀에서 커넥션을 2개 획득하고 반환하지는 않았다. 따라서 풀에 있는 10개의 커넥션 중에 2개를 가지고 있는 상태이다. 그래서 마지막 로그를 보면 사용중인 커넥션 `active=2`, 풀에서 대기 상태인 커넥션 `idle=8`을 확인할 수 있다.

```
MyPool - After adding stats (total=10, active=2, idle=8, waiting=0)
```

> **참고**
>
> HikariCP 커넥션 풀에 대한 더 자세한 내용은 다음 공식 사이트를 참고하자.
>
> https://github.com/brettwooldridge/HikariCP

## 스프링 부트 3.1 이상 - 로그 출력 안되는 문제 해결

히카리 커넥션 풀을 테스트하는 `dataSourceConnectionPool()`을 실행할 때, 스프링 부트 3.1 이상을 사용한다면 전체 로그가 아니라 다음과 같은 간략한 로그만 출력된다.

**스프링 부트 3.1 이상**

```
14:14:55.926 [Test worker] INFO com.zaxxer.hikari.HikariDataSource -- MyPool -
Starting...
14:14:55.993 [Test worker] INFO com.zaxxer.hikari.pool.HikariPool -- MyPool -
Added connection conn0: url=jdbc:h2:tcp://localhost/~/test user=SA
14:14:55.994 [Test worker] INFO com.zaxxer.hikari.HikariDataSource -- MyPool -
Start completed.
14:14:55.997 [Test worker] INFO hello.jdbc.connection.ConnectionTest --
connection=HikariProxyConnection@2066892165 wrapping conn0: url=jdbc:h2:tcp://
```

```
localhost/~/test user=SA, class=class
com.zaxxer.hikari.pool.HikariProxyConnection
14:14:56.000 [Test worker] INFO hello.jdbc.connection.ConnectionTest --
connection=HikariProxyConnection@276869158 wrapping conn1: url=jdbc:h2:tcp://
localhost/~/test user=SA, class=class
com.zaxxer.hikari.pool.HikariProxyConnection
```

이때는 다음 위치에 파일을 만들어서 넣으면 된다.

`src/main/resources/logback.xml`

```
<configuration>

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -%kvp-
%msg%n</pattern>
        </encoder>
    </appender>

    <root level="DEBUG">
        <appender-ref ref="STDOUT" />
    </root>

</configuration>
```

스프링 부트 3.1 부터 기본 로그 레벨을 `INFO`로 빠르게 설정하기 때문에 로그를 확인할 수 없는데, 이렇게하면 기본 로그 레벨을 `DEBUG`로 설정해서 강의 내용과 같이 로그를 확인할 수 있다.

# DataSource 적용

이번에는 애플리케이션에 `DataSource`를 적용해보자.

기존 코드를 유지하기 위해 기존 코드를 복사해서 새로 만들자.
`MemberRepositoryV0` ➜ `MemberRepositoryV1`
`MemberRepositoryV0Test` ➜ `MemberRepositoryV1Test`

### MemberRepositoryV1

```java
package hello.jdbc.repository;

import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;
import org.springframework.jdbc.support.JdbcUtils;

import javax.sql.DataSource;
import java.sql.*;
import java.util.NoSuchElementException;

/**
 * JDBC - DataSource 사용, JdbcUtils 사용
 */
@Slf4j
public class MemberRepositoryV1 {

    private final DataSource dataSource;

    public MemberRepositoryV1(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    //save()...
    //findById()...
    //update()....
    //delete()....

    private void close(Connection con, Statement stmt, ResultSet rs) {
        JdbcUtils.closeResultSet(rs);
        JdbcUtils.closeStatement(stmt);
        JdbcUtils.closeConnection(con);
    }

    private Connection getConnection() throws SQLException {
        Connection con = dataSource.getConnection();
        log.info("get connection={}, class={}", con, con.getClass());
        return con;
    }

}
```

- `DataSource` 의존관계 주입
  - 외부에서 `DataSource`를 주입 받아서 사용한다. 이제 직접 만든 `DBConnectionUtil`을 사용하지 않아도 된다.

    - `DataSource`는 표준 인터페이스 이기 때문에 `DriverManagerDataSource`에서 `HikariDataSource`로 변경되어도 해당 코드를 변경하지 않아도 된다.
- `JdbcUtils` 편의 메서드
    - 스프링은 JDBC를 편리하게 다룰 수 있는 `JdbcUtils`라는 편의 메서드를 제공한다.
    - `JdbcUtils`을 사용하면 커넥션을 좀 더 편리하게 닫을 수 있다.

**MemberRepositoryV1Test**

```
package hello.jdbc.repository;

import com.zaxxer.hikari.HikariDataSource;
import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import java.sql.SQLException;
import java.util.NoSuchElementException;

import static hello.jdbc.connection.ConnectionConst.*;
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatThrownBy;

@Slf4j
class MemberRepositoryV1Test {

    MemberRepositoryV1 repository;

    @BeforeEach
    void beforeEach() throws Exception {
        //기본 DriverManager - 항상 새로운 커넥션 획득
        //DriverManagerDataSource dataSource =
        //        new DriverManagerDataSource(URL, USERNAME, PASSWORD);

        //커넥션 풀링: HikariProxyConnection -> JdbcConnection
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setJdbcUrl(URL);
        dataSource.setUsername(USERNAME);
        dataSource.setPassword(PASSWORD);

        repository = new MemberRepositoryV1(dataSource);
```

```
    }

    @Test
    void crud() throws SQLException, InterruptedException {
        log.info("start");

        //save
        Member member = new Member("memberV0", 10000);
        repository.save(member);

        //findById
        Member memberById = repository.findById(member.getMemberId());
        assertThat(memberById).isNotNull();

        //update: money: 10000 -> 20000
        repository.update(member.getMemberId(), 20000);
        Member updatedMember = repository.findById(member.getMemberId());
        assertThat(updatedMember.getMoney()).isEqualTo(20000);

        //delete
        repository.delete(member.getMemberId());
        assertThatThrownBy(() -> repository.findById(member.getMemberId()))
                .isInstanceOf(NoSuchElementException.class);
    }

}
```

- `MemberRepositoryV1`은 `DataSource` 의존관계 주입이 필요하다.

> **오타 교정**
> 영상에서는 `dataSource.setPassword(PASSWORD)` 대신에 `dataSource.PoolName(PASSWORD)`라는 오타가 입력되었다. 현재 `password`가 없기 때문에 생략할 수 있어서 문제는 없지만, 이 부분은 교재 내용처럼 `dataSource.setPassword(PASSWORD)`를 사용하는 것이 맞다.

**DriverManagerDataSource 사용**

```
get connection=conn0: url=jdbc:h2:.. user=SA class=class
org.h2.jdbc.JdbcConnection
get connection=conn1: url=jdbc:h2:.. user=SA class=class
org.h2.jdbc.JdbcConnection
get connection=conn2: url=jdbc:h2:.. user=SA class=class
org.h2.jdbc.JdbcConnection
get connection=conn3: url=jdbc:h2:.. user=SA class=class
org.h2.jdbc.JdbcConnection
```

```
get connection=conn4: url=jdbc:h2:.. user=SA class=class
org.h2.jdbc.JdbcConnection
get connection=conn5: url=jdbc:h2:.. user=SA class=class
org.h2.jdbc.JdbcConnection
```

- `DriverManagerDataSource`를 사용하면 `conn0~5` 번호를 통해서 항상 새로운 커넥션이 생성되어서 사용되는 것을 확인할 수 있다.

**HikariDataSource 사용**

```
get connection=HikariProxyConnection@xxxxxxxx1 wrapping conn0: url=jdbc:h2:...
user=SA
get connection=HikariProxyConnection@xxxxxxxx2 wrapping conn0: url=jdbc:h2:...
user=SA
get connection=HikariProxyConnection@xxxxxxxx3 wrapping conn0: url=jdbc:h2:...
user=SA
get connection=HikariProxyConnection@xxxxxxxx4 wrapping conn0: url=jdbc:h2:...
user=SA
get connection=HikariProxyConnection@xxxxxxxx5 wrapping conn0: url=jdbc:h2:...
user=SA
get connection=HikariProxyConnection@xxxxxxxx6 wrapping conn0: url=jdbc:h2:...
user=SA
```

- 커넥션 풀 사용시 `conn0` 커넥션이 재사용 된 것을 확인할 수 있다.
- 테스트는 순서대로 실행되기 때문에 커넥션을 사용하고 다시 돌려주는 것을 반복한다. 따라서 `conn0`만 사용된다.
- 웹 애플리케이션에 동시에 여러 요청이 들어오면 여러 쓰레드에서 커넥션 풀의 커넥션을 다양하게 가져가는 상황을 확인할 수 있다.

**DI**

`DriverManagerDataSource` ➜ `HikariDataSource`로 변경해도 `MemberRepositoryV1`의 코드는 전혀 변경하지 않아도 된다. `MemberRepositoryV1`는 `DataSource` 인터페이스에만 의존하기 때문이다. 이것이 `DataSource`를 사용하는 장점이다.(DI + OCP)

# 정리