29. Working with SQL databases

The Spring Framework provides extensive support for working with SQL databases. From direct JDBC access using `JdbcTemplate` to complete 'object relational mapping' technologies such as Hibernate. Spring Data provides an additional level of functionality, creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

29.1 Configure a DataSource

Java's `javax.sql.DataSource` interface provides a standard method of working with database connections. Traditionally a DataSource uses a `URL` along with some credentials to establish a database connection.

29.1.1 Embedded Database Support

It's often convenient to develop applications using an in-memory embedded database. Obviously, in-memory databases do not provide persistent storage; you will need to populate your database when your application starts and be prepared to throw away data when your application ends.

The 'How-to' section includes a *section on how to initialize a database*

Spring Boot can auto-configure embedded H2, HSQL and Derby databases. You don't need to provide any connection URLs, simply include a build dependency to the embedded database that you want to use.

For example, typical POM dependencies would be:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
```

```
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

If, for whatever reason, you do configure the connection URL for an embedded database, care should be taken to ensure that the database's automatic shutdown is disabled. If you're using H2 you should use `DB_CLOSE_ON_EXIT=FALSE` to do so. If you're using HSQLDB, you should ensure that `shutdown=true` is not used. Disabling the database's automatic shutdown allows Spring Boot to control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

You need a dependency on `spring-jdbc` for an embedded database to be auto-configured. In this example it's pulled in transitively via `spring-boot-starter-data-jpa`.

29.1.2 Connection to a production database

Production database connections can also be auto-configured using a pooling `DataSource`. Here's the algorithm for choosing a specific implementation:

- We prefer the Tomcat pooling `DataSource` for its performance and concurrency, so if that is available we always choose it.
- If HikariCP is available we will use it.
- If Commons DBCP is available we will use it, but we don't recommend it in production.
- Lastly, if Commons DBCP2 is available we will use it.

If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` 'starter POMs' you will automatically get a dependency to `tomcat-jdbc`.

You can bypass that algorithm completely and specify the connection pool to use via the `spring.datasource.type` property. Also, additional connection pools can always be configured manually. If you define your own `DataSource` bean, auto-configuration will not occur.

DataSource configuration is controlled by external configuration properties in `spring.datasource.*`. For example, you might declare the following section in `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

You often won't need to specify the `driver-class-name` since Spring boot can deduce it for most databases from the `url`.

For a pooling `DataSource` to be created we need to be able to verify that a valid `Driver` class is available, so we check for that before doing anything. I.e. if you set `spring.datasource.driver-class-name=com.mysql.jdbc.Driver` then that class has to be loadable.

See `DataSourceProperties` for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine tune implementation-specific settings using the `spring.datasource.*` prefix, refer to the documentation of the connection pool implementation you are using for more details.

For instance, if you are using the Tomcat connection pool you could customize many additional settings:

```
# Number of ms to wait before throwing an exception if no connection is available
spring.datasource.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at th
spring.datasource.max-active=50

# Validate the connection before borrowing it from the pool.
spring.datasource.test-on-borrow=true
```

29.1.3 Connection to a JNDI DataSource

If you are deploying your Spring Boot application to an Application Server you might want to

access it using JNDI.

The `spring.datasource.jndi-name` property can be used as an alternative to the `spring.datasource.url`, `spring.datasource.username` and `spring.datasource.password` properties to access the `DataSource` from a specific JNDI location. For example, the following section in `application.properties` shows how you can access a JBoss AS defined `DataSource`:

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

29.2 Using JdbcTemplate

Spring's `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured and you can `@Autowire` them directly into your own beans:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public MyBean(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    // ...

}
```

29.3 JPA and 'Spring Data'

The Java Persistence API is a standard technology that allows you to 'map' objects to relational databases. The `spring-boot-starter-data-jpa` POM provides a quick way to get started. It provides the following key dependencies:

- Hibernate — One of the most popular JPA implementations.

- Spring ORMs — Core ORM support from the Spring Framework.

We won't go into too many details of JPA or Spring Data here. You can follow the 'Accessing Data with JPA' guide from spring.io and read the Spring Data JPA and Hibernate reference documentation.

29.3.1 Entity Classes

Traditionally, JPA 'Entity' classes are specified in a `persistence.xml` file. With Spring Boot this file is not necessary and instead 'Entity Scanning' is used. By default all packages below your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) will be searched.

Any classes annotated with `@Entity`, `@Embeddable` or `@MappedSuperclass` will be considered. A typical entity class would look something like this:

```
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String state;

    // ... additional members, often include @OneToMany mappings

    protected City() {
        // no-args constructor required by JPA spec
        // this one is protected since it shouldn't be used directly
    }
```

```
        this.name = name;
        this.country = country;
    }

    public String getName() {
        return this.name;
    }

    public String getState() {
        return this.state;
    }

    // ... etc

}
```

You can customize entity scanning locations using the `@EntityScan` annotation. See the *Section 73.4, "Separate @Entity definitions from Spring configuration"* how-to.

29.3.2 Spring Data JPA Repositories

Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all cities in a given state.

For more complex queries you can annotate your method using Spring Data's `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you are using auto-configuration, repositories will be searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

Here is a typical Spring Data repository:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;
```

```
    Page<City> findAll(Pageable pageable);

    City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

We have barely scratched the surface of Spring Data JPA. For complete details check their reference documentation.

29.3.3 Creating and dropping JPA databases

By default, JPA databases will be automatically created **only** if you use an embedded database (H2, HSQL or Derby). You can explicitly configure JPA settings using `spring.jpa.*` properties. For example, to create and drop tables you can add the following to your `application.properties`.

```
spring.jpa.hibernate.ddl-auto=create-drop
```

Hibernate's own internal property name for this (if you happen to remember it better) is `hibernate.hbm2ddl.auto`. You can set it, along with other Hibernate native properties, using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). Example:

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

passes `hibernate.globally_quoted_identifiers` to the Hibernate entity manager.

By default the DDL execution (or validation) is deferred until the `ApplicationContext` has started. There is also a `spring.jpa.generate-ddl` flag, but it is not used if Hibernate autoconfig is active because the `ddl-auto` settings are more fine-grained.

29.4 Using H2's web console

The H2 database provides a browser-based console that Spring Boot can auto-configure for you. The console will be auto-configured when the following conditions are met:

- You are developing a web application
- `com.h2database:h2` is on the classpath
- You are using Spring Boot's developer tools

If you are not using Spring Boot's developer tools, but would still like to make use of H2's console, then you can do so by configuring the `spring.h2.console.enabled` property with a value of `true`. The H2 console is only intended for use during development so care should be taken to ensure that `spring.h2.console.enabled` is not set to `true` in production.

29.4.1 Changing the H2 console's path

By default the console will be available at `/h2-console`. You can customize the console's path using the `spring.h2.console.path` property.

29.4.2 Securing the H2 console

When Spring Security is on the classpath and basic auth is enabled, the H2 console will be automatically secured using basic auth. The following properties can be used to customize the security configuration:

- `security.user.role`
- `security.basic.authorize-mode`
- `security.basic.enabled`