SQL Databases

The Spring Framework provides extensive support for working with SQL databases, from direct JDBC access using `JdbcClient` or `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 conven-

Configure a DataSource

Java's `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.

TIP

See the Configure a Custom DataSource section of the "How-to Guides" for more advanced examples, typically to take full control over the configuration of the DataSource.

Embedded Database Support

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

TIP

The "How-to Guides" section includes a section on how to initialize a database.

Spring Boot can auto-configure embedded H2, HSQL, and Derby databases. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use. If there are multiple embedded databases on the classpath, set the `spring.datasource.embedded-database-connection` configuration property to control which one is used. Setting the property to `none` disables auto-configuration of an embedded database.

NOTE

If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.datasource.generate-unique-name` to `true`.

For example, the typical POM dependencies would be as follows:

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

NOTE

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

TIP

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

Connection to a Production Database

Production database connections can also be auto-configured by using a pooling `DataSource`.

DataSource Configuration

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

Properties YAML

```properties
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
```

NOTE

You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database.

TIP

Spring Boot can deduce the JDBC driver class for most databases from the URL. If you need to specify a specific class, you can use the `spring.datasource.driver-class-name` property.

Edit this Page
GitHub Project
Stack Overflow

NOTE

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. In other words, if you set `spring.datasource.driver-class-name=com.mysql.jdbc.Driver`, then that class has to be loadable.

See `DataSourceProperties` API documentation 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 by using their respective prefix (`spring.datasource.hikari.*`, `spring.datasource.tomcat.*`, `spring.datasource.dbcp2.*`, and `spring.datasource.oracleucp.*`). See the documentation of the connection pool implementation you are using for more details.

For instance, if you use the Tomcat connection pool, you could customize many additional settings, as shown in the following example:

Properties | YAML

PROPERTIES

```
spring.datasource.tomcat.max-wait=10000
spring.datasource.tomcat.max-active=50
spring.datasource.tomcat.test-on-borrow=true
```

This will set the pool to wait 10000ms before throwing an exception if no connection is available, limit the maximum number of connections to 50 and validate the connection before borrowing it from the pool.

Supported Connection Pools

Spring Boot uses the following algorithm for choosing a specific implementation:

1. We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it.
2. Otherwise, if the Tomcat pooling `DataSource` is available, we use it.
3. Otherwise, if Commons DBCP2 is available, we use it.
4. If none of HikariCP, Tomcat, and DBCP2 are available and if Oracle UCP is available, we use it.

NOTE

If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` starters, you automatically get a dependency to HikariCP.

You can bypass that algorithm completely and specify the connection pool to use by setting the `spring.datasource.type` property. This is especially important if you run your application in a Tomcat container, as `tomcat-jdbc` is provided by default.

Additional connection pools can always be configured manually, using `DataSourceBuilder`. If you define your own `DataSource` bean, auto-configuration does not occur. The following connection pools are supported by `DataSourceBuilder`:

- HikariCP
- Tomcat pooling `DataSource`
- Commons DBCP2
- Oracle UCP & `OracleDataSource`
- Spring Framework's `SimpleDriverDataSource`
- H2 `JdbcDataSource`
- PostgreSQL `PGSimpleDataSource`
- C3P0
- Vibur

Connection to a JNDI DataSource

If you deploy your Spring Boot application to an Application Server, you might want to configure and manage your DataSource by using your Application Server's built-in features and access it by 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`:

Properties | YAML

PROPERTIES

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

Using JdbcTemplate

Spring's `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured, and you can autowire them directly into your own beans, as shown in the following example:

Java | Kotlin

JAVA

```
@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

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

```
    public void doSomething() {
        this.jdbcTemplate ...
    }

}
```

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, as shown in the following example:

Properties | YAML

```
spring.jdbc.template.max-rows=500
```

If tuning of SQL exceptions is required, you can define your own `SQLExceptionTranslator` bean so that it is associated with the auto-configured `JdbcTemplate`.

NOTE

The `NamedParameterJdbcTemplate` reuses the same `JdbcTemplate` instance behind the scenes. If more than one `JdbcTemplate` is defined and no primary candidate exists, the `NamedParameterJdbcTemplate` is not auto-configured.

Using JdbcClient

Spring's `JdbcClient` is auto-configured based on the presence of a `NamedParameterJdbcTemplate`. You can inject it directly in your own beans as well, as shown in the following example:

Java | Kotlin

```
@Component
public class MyBean {

    private final JdbcClient jdbcClient;

    public MyBean(JdbcClient jdbcClient) {
        this.jdbcClient = jdbcClient;
    }

    public void doSomething() {
        this.jdbcClient ...
    }

}
```

If you rely on auto-configuration to create the underlying `JdbcTemplate`, any customization using `spring.jdbc.template.*` properties is taken into account in the client as well.

JPA and Spring Data JPA

The Java Persistence API is a standard technology that lets you "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 Data JPA: Helps you to implement JPA-based repositories.
- Spring ORM: Core ORM support from the Spring Framework.

TIP

We do not 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.

Entity Classes

Traditionally, JPA "Entity" classes are specified in a `persistence.xml` file. With Spring Boot, this file is not necessary and "Entity Scanning" is used instead. By default the auto-configuration packages are scanned.

Any classes annotated with `@Entity`, `@Embeddable`, or `@MappedSuperclass` are considered. A typical entity class resembles the following example:

Java | Kotlin

```
@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 should not be used directly
    }

    public City(String name, String state) {
        this.name = name;
```

```
        this.state = state;
    }

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

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

    // ... etc

}
```

TIP

You can customize entity scanning locations by using the `@EntityScan` annotation. See the Separate @Entity Definitions from Spring Configuration section of the "How-to Guides".

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 the cities in a given state.

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

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use auto-configuration, the auto-configuration packages are searched for repositories.

TIP

You can customize the locations to look for repositories using `@EnableJpaRepositories`.

The following example shows a typical Spring Data repository interface definition:

Java Kotlin

```
JAVA
public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

Spring Data JPA repositories support three different modes of bootstrapping: default, deferred, and lazy. To enable deferred or lazy bootstrapping, set the `spring.data.jpa.repositories.bootstrap-mode` property to `deferred` or `lazy` respectively. When using deferred or lazy bootstrapping, the auto-configured `EntityManagerFactoryBuilder` will use the context's `AsyncTaskExecutor`, if any, as the bootstrap executor. If more than one exists, the one named `applicationTaskExecutor` will be used.

NOTE

When using deferred or lazy bootstrapping, make sure to defer any access to the JPA infrastructure after the application context bootstrap phase. You can use `SmartInitializingSingleton` to invoke any initialization that requires the JPA infrastructure. For JPA components (such as converters) that are created as Spring beans, use `ObjectProvider` to delay the resolution of dependencies, if any.

TIP

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

Spring Data Envers Repositories

If Spring Data Envers is available, JPA repositories are auto-configured to support typical Envers queries.

To use Spring Data Envers, make sure your repository extends from `RevisionRepository` as shown in the following example:

Java Kotlin

```
JAVA
public interface CountryRepository extends RevisionRepository<Country, Long, Integer>, Repositor

    Page<Country> findAll(Pageable pageable);

}
```

NOTE

For more details, check the Spring Data Envers reference documentation.

Creating and Dropping JPA Databases

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

Properties YAML

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

NOTE

NOTE

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, by using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). The following line shows an example of setting JPA properties for Hibernate:

Properties | YAML

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

The line in the preceding example passes a value of `true` for the `hibernate.globally_quoted_identifiers` property to the Hibernate entity manager.

By default, the DDL execution (or validation) is deferred until the `ApplicationContext` has started.

Open EntityManager in View

If you are running a web application, Spring Boot by default registers `OpenEntityManagerInViewInterceptor` to apply the "Open EntityManager in View" pattern, to allow for lazy loading in web views. If you do not want this behavior, you should set `spring.jpa.open-in-view` to `false` in your `application.properties`.

Spring Data JDBC

Spring Data includes repository support for JDBC and will automatically generate SQL for the methods on `CrudRepository`. For more advanced queries, a `@Query` annotation is provided.

Spring Boot will auto-configure Spring Data's JDBC repositories when the necessary dependencies are on the classpath. They can be added to your project with a single dependency on `spring-boot-starter-data-jdbc`. If necessary, you can take control of Spring Data JDBC's configuration by adding the `@EnableJdbcRepositories` annotation or an `AbstractJdbcConfiguration` subclass to your application.

TIP

For complete details of Spring Data JDBC, see the reference documentation.

Using H2's Web Console

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

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

TIP

If you are not using Spring Boot's developer tools but would still like to make use of H2's console, you can configure the `spring.h2.console.enabled` property with a value of `true`.

NOTE

The H2 console is only intended for use during development, so you should take care to ensure that `spring.h2.console.enabled` is not set to `true` in production.

Changing the H2 Console's Path

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

Accessing the H2 Console in a Secured Application

H2 Console uses frames and, as it is intended for development only, does not implement CSRF protection measures. If your application uses Spring Security, you need to configure it to

- disable CSRF protection for requests against the console,
- set the header `X-Frame-Options` to `SAMEORIGIN` on responses from the console.

More information on CSRF and the header X-Frame-Options can be found in the Spring Security Reference Guide.

In simple setups, a `SecurityFilterChain` like the following can be used:

Java | Kotlin

```
JAVA
@Profile("dev")
@Configuration(proxyBeanMethods = false)
public class DevProfileSecurityConfiguration {

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    SecurityFilterChain h2ConsoleSecurityFilterChain(HttpSecurity http) throws Exception {
        http.securityMatcher(PathRequest.toH2Console());
        http.authorizeHttpRequests(yourCustomAuthorization());
        http.csrf(CsrfConfigurer::disable);
        http.headers((headers) -> headers.frameOptions(FrameOptionsConfig::sameOrigin));
        return http.build();
    }


}
```

WARNING

The H2 console is only intended for use during development. In production, disabling CSRF protection or allowing frames for a

website may create severe security risks.

TIP

`PathRequest.toH2Console()` returns the correct request matcher also when the console's path has been customized.

Using jOOQ

jOOQ Object Oriented Querying (jOOQ) is a popular product from Data Geekery which generates Java code from your database and lets you build type-safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the jOOQ user manual. If you use the `jooq-codegen-maven` plugin and you also use the `spring-boot-starter-parent` "parent POM", you can safely omit the plugin's `<version>` tag. You can also use Spring Boot-defined version variables (such as `h2.version`) to declare the plugin's database dependency. The following listing shows an example:

```xml
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
    <executions>
        ...
    </executions>
    <dependencies>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>${h2.version}</version>
        </dependency>
    </dependencies>
    <configuration>
        <jdbc>
            <driver>org.h2.Driver</driver>
            <url>jdbc:h2:~/yourdatabase</url>
        </jdbc>
        <generator>
            ...
        </generator>
    </configuration>
</plugin>
```

Using DSLContext

The fluent API offered by jOOQ is initiated through the `DSLContext` interface. Spring Boot auto-configures a `DSLContext` as a Spring Bean and connects it to your application `DataSource`. To use the `DSLContext`, you can inject it, as shown in the following example:

Java Kotlin

```java
@Component
public class MyBean {

    private final DSLContext create;

    public MyBean(DSLContext dslContext) {
        this.create = dslContext;
    }


}
```

TIP

The jOOQ manual tends to use a variable named `create` to hold the `DSLContext`.

You can then use the `DSLContext` to construct your queries, as shown in the following example:

Java Kotlin

```java
    public List<GregorianCalendar> authorsBornAfter1980() {
        return this.create.selectFrom(AUTHOR)
            .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCalendar(1980, 0, 1)))
            .fetch(AUTHOR.DATE_OF_BIRTH);
```

jOOQ SQL Dialect

Unless the `spring.jooq.sql-dialect` property has been configured, Spring Boot determines the SQL dialect to use for your datasource. If Spring Boot could not detect the dialect, it uses `DEFAULT`.

NOTE

Spring Boot can only auto-configure dialects supported by the open source version of jOOQ.

Customizing jOOQ

More advanced customizations can be achieved by defining your own `DefaultConfigurationCustomizer` bean that will be invoked prior to creating the `Configuration` `@Bean`. This takes precedence to anything that is applied by the auto-configuration.

You can also create your own `Configuration` `@Bean` if you want to take complete control of the jOOQ configuration.

Using R2DBC

The Reactive Relational Database Connectivity (R2DBC) project brings reactive programming APIs to relational databases. R2DBC's `Connection` provides a standard method of working with non-blocking database connections. Connections are provided by using a `ConnectionFactory`, similar to a `DataSource` with jdbc.

`ConnectionFactory` configuration is controlled by external configuration properties in `spring.r2dbc.*`. For example, you might declare the following section in `application.properties`:

Properties | YAML

```
PROPERTIES
spring.r2dbc.url=r2dbc:postgresql://localhost/test
spring.r2dbc.username=dbuser
spring.r2dbc.password=dbpass
```

TIP

You do not need to specify a driver class name, since Spring Boot obtains the driver from R2DBC's Connection Factory discovery.

NOTE

At least the url should be provided. Information specified in the URL takes precedence over individual properties, that is `name`, `username`, `password` and pooling options.

TIP

The "How-to Guides" section includes a section on how to initialize a database.

To customize the connections created by a `ConnectionFactory`, that is, set specific parameters that you do not want (or cannot) configure in your central database configuration, you can use a `ConnectionFactoryOptionsBuilderCustomizer` `@Bean`. The following example shows how to manually override the database port while the rest of the options are taken from the application configuration:

Java | Kotlin

```
JAVA
@Configuration(proxyBeanMethods = false)
public class MyR2dbcConfiguration {

    @Bean
    public ConnectionFactoryOptionsBuilderCustomizer connectionFactoryPortCustomizer() {
        return (builder) -> builder.option(ConnectionFactoryOptions.PORT, 5432);
    }

}
```

The following examples show how to set some PostgreSQL connection options:

Java | Kotlin

```
JAVA
@Configuration(proxyBeanMethods = false)
public class MyPostgresR2dbcConfiguration {

    @Bean
    public ConnectionFactoryOptionsBuilderCustomizer postgresCustomizer() {
        Map<String, String> options = new HashMap<>();
        options.put("lock_timeout", "30s");
        options.put("statement_timeout", "60s");
        return (builder) -> builder.option(PostgresqlConnectionFactoryProvider.OPTIONS, options)
    }

}
```

When a `ConnectionFactory` bean is available, the regular JDBC `DataSource` auto-configuration backs off. If you want to retain the JDBC `DataSource` auto-configuration, and are comfortable with the risk of using the blocking JDBC API in a reactive application, add `@Import(DataSourceAutoConfiguration.class)` on a `@Configuration` class in your application to re-enable it.

Embedded Database Support

Similarly to the JDBC support, Spring Boot can automatically configure an embedded database for reactive usage. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use, as shown in the following example:

```
XML
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

NOTE

If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.r2dbc.generate-unique-name` to `true`.

Using DatabaseClient

A `DatabaseClient` bean is auto-configured, and you can autowire it directly into your own beans, as shown in the following example:

Java Kotlin

```java
@Component
public class MyBean {

    private final DatabaseClient databaseClient;

    public MyBean(DatabaseClient databaseClient) {
        this.databaseClient = databaseClient;
    }

    // ...

}
```

Spring Data R2DBC Repositories

Spring Data R2DBC repositories are interfaces that you can define to access data. Queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all the cities in a given state.

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

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use auto-configuration, the auto-configuration packages are searched for repositories.

The following example shows a typical Spring Data repository interface definition:

Java Kotlin

```java
public interface CityRepository extends Repository<City, Long> {

    Mono<City> findByNameAndStateAllIgnoringCase(String name, String state);

```