*Properties*

```
spring.security.saml2.relyingparty.registration.my-relying-
party1.signing.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party1.signing.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party1.decryption.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party1.decryption.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party1.singlelogout.url=https://myapp/logout/saml2/slo
spring.security.saml2.relyingparty.registration.my-relying-
party1.singlelogout.response-url=https://remoteidp2.slo.url
spring.security.saml2.relyingparty.registration.my-relying-
party1.singlelogout.binding=POST
spring.security.saml2.relyingparty.registration.my-relying-
party1.assertingparty.verification.credentials[0].certificate-location=path-to-
verification-cert
spring.security.saml2.relyingparty.registration.my-relying-
party1.assertingparty.entity-id=remote-idp-entity-id1
spring.security.saml2.relyingparty.registration.my-relying-party1.assertingparty.sso-
url=https://remoteidp1.sso.url

spring.security.saml2.relyingparty.registration.my-relying-
party2.signing.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party2.signing.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party2.decryption.credentials[0].private-key-location=path-to-private-key
spring.security.saml2.relyingparty.registration.my-relying-
party2.decryption.credentials[0].certificate-location=path-to-certificate
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.verification.credentials[0].certificate-location=path-to-other-
verification-cert
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.entity-id=remote-idp-entity-id2
spring.security.saml2.relyingparty.registration.my-relying-party2.assertingparty.sso-
url=https://remoteidp2.sso.url
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.singlelogout.url=https://remoteidp2.slo.url
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.singlelogout.response-url=https://myapp/logout/saml2/slo
spring.security.saml2.relyingparty.registration.my-relying-
party2.assertingparty.singlelogout.binding=POST
```

*Yaml*

```
spring:
  security:
    saml2:
      relyingparty:
        registration:
          my-relying-party1:
            signing:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            decryption:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            singlelogout:
               url: "https://myapp/logout/saml2/slo"
               response-url: "https://remoteidp2.slo.url"
               binding: "POST"
            assertingparty:
              verification:
                credentials:
                - certificate-location: "path-to-verification-cert"
              entity-id: "remote-idp-entity-id1"
              sso-url: "https://remoteidp1.sso.url"

          my-relying-party2:
            signing:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            decryption:
              credentials:
              - private-key-location: "path-to-private-key"
                certificate-location: "path-to-certificate"
            assertingparty:
              verification:
                credentials:
                - certificate-location: "path-to-other-verification-cert"
              entity-id: "remote-idp-entity-id2"
              sso-url: "https://remoteidp2.sso.url"
              singlelogout:
                url: "https://remoteidp2.slo.url"
                response-url: "https://myapp/logout/saml2/slo"
                binding: "POST"
```

For SAML2 logout, by default, Spring Security's `Saml2LogoutRequestFilter` and `Saml2LogoutResponseFilter` only process URLs matching `/logout/saml2/slo`. If you want to customize the `url` to which AP-initiated logout requests get sent to or the `response-url` to which an AP sends

logout responses to, to use a different pattern, you need to provide configuration to process that custom pattern. For example, for servlet applications, you can add your own `SecurityFilterChain` that resembles the following:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration(proxyBeanMethods = false)
public class MySamlRelyingPartyConfiguration {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http.authorizeHttpRequests((requests) ->
requests.anyRequest().authenticated());
        http.saml2Login(withDefaults());
        http.saml2Logout((saml2) -> saml2.logoutRequest((request) ->
request.logoutUrl("/SLOService.saml2"))
            .logoutResponse((response) -> response.logoutUrl("/SLOService.saml2")));
        return http.build();
    }

}
```

## 8.5. Spring Session

Spring Boot provides Spring Session auto-configuration for a wide range of data stores. When building a servlet web application, the following stores can be auto-configured:

- Redis
- JDBC
- Hazelcast
- MongoDB

Additionally, Spring Boot for Apache Geode provides auto-configuration for using Apache Geode as a session store.

The servlet auto-configuration replaces the need to use `@Enable*HttpSession`.

If a single Spring Session module is present on the classpath, Spring Boot uses that store implementation automatically. If you have more than one implementation, Spring Boot uses the following order for choosing a specific implementation:

1. Redis

2. JDBC
3. Hazelcast
4. MongoDB
5. If none of Redis, JDBC, Hazelcast and MongoDB are available, we do not configure a `SessionRepository`.

When building a reactive web application, the following stores can be auto-configured:

- Redis
- MongoDB

The reactive auto-configuration replaces the need to use `@Enable*WebSession`.

Similar to the servlet configuration, if you have more than one implementation, Spring Boot uses the following order for choosing a specific implementation:

1. Redis
2. MongoDB
3. If neither Redis nor MongoDB are available, we do not configure a `ReactiveSessionRepository`.

Each store has specific additional settings. For instance, it is possible to customize the name of the table for the JDBC store, as shown in the following example:

*Properties*

```
spring.session.jdbc.table-name=SESSIONS
```

*Yaml*

```
spring:
  session:
    jdbc:
      table-name: "SESSIONS"
```

For setting the timeout of the session you can use the `spring.session.timeout` property. If that property is not set with a servlet web application, the auto-configuration falls back to the value of `server.servlet.session.timeout`.

You can take control over Spring Session's configuration using `@Enable*HttpSession` (servlet) or `@Enable*WebSession` (reactive). This will cause the auto-configuration to back off. Spring Session can then be configured using the annotation's attributes rather than the previously described configuration properties.

## 8.6. Spring for GraphQL

If you want to build GraphQL applications, you can take advantage of Spring Boot's auto-configuration for Spring for GraphQL. The Spring for GraphQL project is based on GraphQL Java.

You'll need the `spring-boot-starter-graphql` starter at a minimum. Because GraphQL is transport-agnostic, you'll also need to have one or more additional starters in your application to expose your GraphQL API over the web:

| Starter | Transport | Implementation |
|---|---|---|
| `spring-boot-starter-web` | HTTP | Spring MVC |
| `spring-boot-starter-websocket` | WebSocket | WebSocket for Servlet apps |
| `spring-boot-starter-webflux` | HTTP, WebSocket | Spring WebFlux |
| `spring-boot-starter-rsocket` | TCP, WebSocket | Spring WebFlux on Reactor Netty |

### 8.6.1. GraphQL Schema

A Spring GraphQL application requires a defined schema at startup. By default, you can write ".graphqls" or ".gqls" schema files under `src/main/resources/graphql/**` and Spring Boot will pick them up automatically. You can customize the locations with `spring.graphql.schema.locations` and the file extensions with `spring.graphql.schema.file-extensions`.

| NOTE | If you want Spring Boot to detect schema files in all your application modules and dependencies for that location, you can set `spring.graphql.schema.locations` to `"classpath*:graphql/**/"` (note the `classpath*:` prefix). |
|---|---|

In the following sections, we'll consider this sample GraphQL schema, defining two types and two queries:

```
type Query {
    greeting(name: String! = "Spring"): String!
    project(slug: ID!): Project
}

""" A Project in the Spring portfolio """
type Project {
    """ Unique string id used in URLs """
    slug: ID!
    """ Project name """
    name: String!
    """ URL of the git repository """
    repositoryUrl: String!
    """ Current support status """
    status: ProjectStatus!
}

enum ProjectStatus {
    """ Actively supported by the Spring team """
    ACTIVE
    """ Supported by the community """
    COMMUNITY
    """ Prototype, not officially supported yet  """
    INCUBATING
    """ Project being retired, in maintenance mode """
    ATTIC
    """ End-Of-Lifed """
    EOL
}
```

**NOTE** By default, field introspection will be allowed on the schema as it is required for tools such as GraphiQL. If you wish to not expose information about the schema, you can disable introspection by setting `spring.graphql.schema.introspection.enabled` to `false`.

### 8.6.2. GraphQL RuntimeWiring

The GraphQL Java `RuntimeWiring.Builder` can be used to register custom scalar types, directives, type resolvers, `DataFetcher`, and more. You can declare `RuntimeWiringConfigurer` beans in your Spring config to get access to the `RuntimeWiring.Builder`. Spring Boot detects such beans and adds them to the GraphQlSource builder.

Typically, however, applications will not implement `DataFetcher` directly and will instead create annotated controllers. Spring Boot will automatically detect `@Controller` classes with annotated handler methods and register those as `DataFetcher`s. Here's a sample implementation for our greeting query with a `@Controller` class:

*Java*

```
import org.springframework.graphql.data.method.annotation.Argument;
import org.springframework.graphql.data.method.annotation.QueryMapping;
import org.springframework.stereotype.Controller;

@Controller
public class GreetingController {

    @QueryMapping
    public String greeting(@Argument String name) {
        return "Hello, " + name + "!";
    }

}
```

*Kotlin*

```
import org.springframework.graphql.data.method.annotation.Argument
import org.springframework.graphql.data.method.annotation.QueryMapping
import org.springframework.stereotype.Controller

@Controller
class GreetingController {

    @QueryMapping
    fun greeting(@Argument name: String): String {
        return "Hello, $name!"
    }

}
```

### 8.6.3. Querydsl and QueryByExample Repositories Support

Spring Data offers support for both Querydsl and QueryByExample repositories. Spring GraphQL can configure Querydsl and QueryByExample repositories as `DataFetcher`.

Spring Data repositories annotated with `@GraphQlRepository` and extending one of:

- `QuerydslPredicateExecutor`
- `ReactiveQuerydslPredicateExecutor`
- `QueryByExampleExecutor`
- `ReactiveQueryByExampleExecutor`

are detected by Spring Boot and considered as candidates for `DataFetcher` for matching top-level queries.

### 8.6.4. Transports

#### HTTP and WebSocket

The GraphQL HTTP endpoint is at HTTP POST `/graphql` by default. The path can be customized with `spring.graphql.path`.

| TIP | The HTTP endpoint for both Spring MVC and Spring WebFlux is provided by a `RouterFunction` bean with an `@Order` of `0`. If you define your own `RouterFunction` beans, you may want to add appropriate `@Order` annotations to ensure that they are sorted correctly. |
|---|---|

The GraphQL WebSocket endpoint is off by default. To enable it:

- For a Servlet application, add the WebSocket starter `spring-boot-starter-websocket`
- For a WebFlux application, no additional dependency is required
- For both, the `spring.graphql.websocket.path` application property must be set

Spring GraphQL provides a Web Interception model. This is quite useful for retrieving information from an HTTP request header and set it in the GraphQL context or fetching information from the same context and writing it to a response header. With Spring Boot, you can declare a `WebInterceptor` bean to have it registered with the web transport.

Spring MVC and Spring WebFlux support CORS (Cross-Origin Resource Sharing) requests. CORS is a critical part of the web config for GraphQL applications that are accessed from browsers using different domains.

Spring Boot supports many configuration properties under the `spring.graphql.cors.*` namespace; here's a short configuration sample:

*Properties*

```
spring.graphql.cors.allowed-origins=https://example.org
spring.graphql.cors.allowed-methods=GET,POST
spring.graphql.cors.max-age=1800s
```

*Yaml*

```
spring:
  graphql:
    cors:
      allowed-origins: "https://example.org"
      allowed-methods: GET,POST
      max-age: 1800s
```

#### RSocket

RSocket is also supported as a transport, on top of WebSocket or TCP. Once the RSocket server is

configured, we can configure our GraphQL handler on a particular route using `spring.graphql.rsocket.mapping`. For example, configuring that mapping as `"graphql"` means we can use that as a route when sending requests with the `RSocketGraphQlClient`.

Spring Boot auto-configures a `RSocketGraphQlClient.Builder<?>` bean that you can inject in your components:

*Java*

```
@Component
public class RSocketGraphQlClientExample {

    private final RSocketGraphQlClient graphQlClient;

    public RSocketGraphQlClientExample(RSocketGraphQlClient.Builder<?> builder) {
        this.graphQlClient = builder.tcp("example.spring.io",
8181).route("graphql").build();
    }
```

*Kotlin*

```
@Component
class RSocketGraphQlClientExample(private val builder:
RSocketGraphQlClient.Builder<*>) {
```

And then send a request:

*Java*

```
Mono<Book> book = this.graphQlClient.document("{ bookById(id: \"book-1\"){ id name
pageCount author } }")
    .retrieve("bookById")
    .toEntity(Book.class);
```

*Kotlin*

```
val book = graphQlClient.document(
    """
    {
        bookById(id: "book-1"){
            id
            name
            pageCount
            author
        }
    }
    """
)
    .retrieve("bookById").toEntity(Book::class.java)
```

### 8.6.5. Exception Handling

Spring GraphQL enables applications to register one or more Spring `DataFetcherExceptionResolver` components that are invoked sequentially. The Exception must be resolved to a list of `graphql.GraphQLError` objects, see Spring GraphQL exception handling documentation. Spring Boot will automatically detect `DataFetcherExceptionResolver` beans and register them with the `GraphQlSource.Builder`.

### 8.6.6. GraphiQL and Schema printer

Spring GraphQL offers infrastructure for helping developers when consuming or developing a GraphQL API.

Spring GraphQL ships with a default GraphiQL page that is exposed at `"/graphiql"` by default. This page is disabled by default and can be turned on with the `spring.graphql.graphiql.enabled` property. Many applications exposing such a page will prefer a custom build. A default implementation is very useful during development, this is why it is exposed automatically with `spring-boot-devtools` during development.

You can also choose to expose the GraphQL schema in text format at `/graphql/schema` when the `spring.graphql.schema.printer.enabled` property is enabled.

## 8.7. Spring HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications, including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that doing so disables the `ObjectMapper` customization described earlier.

| | |
|---|---|
| **WARNING** | `spring-boot-starter-hateoas` is specific to Spring MVC and should not be combined with Spring WebFlux. In order to use Spring HATEOAS with Spring WebFlux, you can add a direct dependency on `org.springframework.hateoas:spring-hateoas` along with `spring-boot-starter-webflux`. |

By default, requests that accept `application/json` will receive an `application/hal+json` response. To disable this behavior set `spring.hateoas.use-hal-as-default-json-media-type` to `false` and define a `HypermediaMappingInformation` or `HalConfiguration` to configure Spring HATEOAS to meet the needs of your application and its clients.

## 8.8. What to Read Next

You should now have a good understanding of how to develop web applications with Spring Boot. The next few sections describe how Spring Boot integrates with various data technologies, messaging systems, and other IO capabilities. You can pick any of these based on your application's needs.

# Chapter 9. Data

Spring Boot integrates with a number of data technologies, both SQL and NoSQL.

## 9.1. 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 conventions to generate queries from your method names.

### 9.1.1. 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.

> **TIP** See the "How-to" section 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" 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:

```
<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*

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

*Yaml*

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/test"
    username: "dbuser"
    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. |
| **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` 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*

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

*Yaml*

```
spring:
  datasource:
    tomcat:
      max-wait: 10000
      max-active: 50
      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

#### 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*

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

*Yaml*

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

### 9.1.2. 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*

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JdbcTemplate jdbcTemplate;

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

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

}
```

*Kotlin*

```
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val jdbcTemplate: JdbcTemplate) {

    fun doSomething() {
        jdbcTemplate.execute("delete from customer")
    }

}
```

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

*Properties*

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

*Yaml*

```
spring:
  jdbc:
    template:
      max-rows: 500
```

| 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. |
|---|---|

### 9.1.3. 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*

```
import org.springframework.jdbc.core.simple.JdbcClient;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JdbcClient jdbcClient;

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

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

}
```

*Kotlin*

```
import org.springframework.jdbc.core.simple.JdbcClient
import org.springframework.stereotype.Component

@Component
class MyBean(private val jdbcClient: JdbcClient) {

    fun doSomething() {
        jdbcClient.sql("delete from customer").update()
    }

}
```

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.

### 9.1.4. 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*

```
import java.io.Serializable;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;

@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

}
```

*Kotlin*

```
import jakarta.persistence.Column
import jakarta.persistence.Entity
import jakarta.persistence.GeneratedValue
import jakarta.persistence.Id
import java.io.Serializable

@Entity
class City : Serializable {

    @Id
    @GeneratedValue
    private val id: Long? = null

    @Column(nullable = false)
    var name: String? = null
        private set

    // ... etc
    @Column(nullable = false)
    var state: String? = null
        private set

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

    protected constructor() {
        // no-args constructor required by JPA spec
        // this one is protected since it should not be used directly
    }

    constructor(name: String?, state: String?) {
        this.name = name
        this.state = state
    }

}
```

**TIP** You can customize entity scanning locations by using the `@EntityScan` annotation. See the "Separate @Entity Definitions from Spring Configuration" how-to.

### 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*

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.City;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

*Kotlin*

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.City
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.repository.Repository

interface CityRepository : Repository<City?, Long?> {

    fun findAll(pageable: Pageable?): Page<City?>?

    fun findByNameAndStateAllIgnoringCase(name: String?, state: String?): City?

}
```

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*

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.Country;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.history.RevisionRepository;

public interface CountryRepository extends RevisionRepository<Country, Long, Integer>,
Repository<Country, Long> {

    Page<Country> findAll(Pageable pageable);

}
```

*Kotlin*

```
import org.springframework.boot.docs.data.sql.jpaandspringdata.entityclasses.Country
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.repository.Repository
import org.springframework.data.repository.history.RevisionRepository

interface CountryRepository :
        RevisionRepository<Country?, Long?, Int>,
        Repository<Country?, Long?> {

    fun findAll(pageable: Pageable?): Page<Country?>?

}
```

**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*

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

*Yaml*

```
spring:
  jpa:
    hibernate.ddl-auto: "create-drop"
```

**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*

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

*Yaml*

```
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`.

### 9.1.5. 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. |

### 9.1.6. 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*

```
import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configurers.CsrfConfigurer;
import
org.springframework.security.config.annotation.web.configurers.HeadersConfigurer.Frame
OptionsConfig;
import org.springframework.security.web.SecurityFilterChain;

@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();
    }


}
```

*Kotlin*

```
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile
import org.springframework.core.Ordered
import org.springframework.core.annotation.Order
import org.springframework.security.config.Customizer
import org.springframework.security.config.annotation.web.builders.HttpSecurity
import org.springframework.security.web.SecurityFilterChain

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

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    fun h2ConsoleSecurityFilterChain(http: HttpSecurity): SecurityFilterChain {
        return http.authorizeHttpRequests(yourCustomAuthorization())
            .csrf { csrf -> csrf.disable() }
            .headers { headers -> headers.frameOptions { frameOptions ->
frameOptions.sameOrigin() } }
            .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. |

### 9.1.7. 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:

```
<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 `org.jooq.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*

```
import java.util.GregorianCalendar;
import java.util.List;

import org.jooq.DSLContext;

import org.springframework.stereotype.Component;

import static org.springframework.boot.docs.data.sql.jooq.dslcontext.Tables.AUTHOR;

@Component
public class MyBean {

    private final DSLContext create;

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


}
```

*Kotlin*

```
import org.jooq.DSLContext
import org.springframework.stereotype.Component
import java.util.GregorianCalendar

@Component
class MyBean(private val 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*

```
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);
```

*Kotlin*

```
fun authorsBornAfter1980(): List<GregorianCalendar> {
    return create.selectFrom<Tables.TAuthorRecord>(Tables.AUTHOR)
        .where(Tables.AUTHOR?.DATE_OF_BIRTH?.greaterThan(GregorianCalendar(1980, 0,
1)))
        .fetch(Tables.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 `org.jooq.Configuration @Bean`. This takes precedence to anything that is applied by the auto-configuration.

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

### 9.1.8. Using R2DBC

The Reactive Relational Database Connectivity (R2DBC) project brings reactive programming APIs to relational databases. R2DBC's `io.r2dbc.spi.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*

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

*Yaml*

```
spring:
  r2dbc:
    url: "r2dbc:postgresql://localhost/test"
    username: "dbuser"
    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" 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*

```
import io.r2dbc.spi.ConnectionFactoryOptions;

import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyR2dbcConfiguration {

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

}
```

*Kotlin*

```
import io.r2dbc.spi.ConnectionFactoryOptions
import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyR2dbcConfiguration {

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

}
```

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

*Java*

```
import java.util.HashMap;
import java.util.Map;

import io.r2dbc.postgresql.PostgresqlConnectionFactoryProvider;

import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@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);
    }

}
```

*Kotlin*

```
import io.r2dbc.postgresql.PostgresqlConnectionFactoryProvider
import
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryOptionsBuilderCustomizer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyPostgresR2dbcConfiguration {

    @Bean
    fun postgresCustomizer(): ConnectionFactoryOptionsBuilderCustomizer {
        val options: MutableMap<String, String> = HashMap()
        options["lock_timeout"] = "30s"
        options["statement_timeout"] = "60s"
        return ConnectionFactoryOptionsBuilderCustomizer { 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:

```
<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*

```
import java.util.Map;

import reactor.core.publisher.Flux;

import org.springframework.r2dbc.core.DatabaseClient;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final DatabaseClient databaseClient;

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

    public Flux<Map<String, Object>> someMethod() {
        return this.databaseClient.sql("select * from user").fetch().all();
    }

}
```

*Kotlin*

```
import org.springframework.r2dbc.core.DatabaseClient
import org.springframework.stereotype.Component
import reactor.core.publisher.Flux

@Component
class MyBean(private val databaseClient: DatabaseClient) {

    fun someMethod(): Flux<Map<String, Any>> {
        return databaseClient.sql("select * from user").fetch().all()
    }

}
```

### 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*

```
import reactor.core.publisher.Mono;

import org.springframework.data.repository.Repository;

public interface CityRepository extends Repository<City, Long> {

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

}
```

*Kotlin*

```
import org.springframework.data.repository.Repository
import reactor.core.publisher.Mono

interface CityRepository : Repository<City?, Long?> {

    fun findByNameAndStateAllIgnoringCase(name: String?, state: String?): Mono<City?>?

}
```

**TIP** We have barely scratched the surface of Spring Data R2DBC. For complete details, see the Spring Data R2DBC reference documentation.

## 9.2. Working with NoSQL Technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies, including:

- Cassandra
- Couchbase
- Elasticsearch
- GemFire or Geode
- LDAP
- MongoDB
- Neo4J
- Redis

Of these, Spring Boot provides auto-configuration for Cassandra, Couchbase, Elasticsearch, LDAP, MongoDB, Neo4J and Redis. Additionally, Spring Boot for Apache Geode provides auto-configuration for Apache Geode. You can make use of the other projects, but you must configure them yourself. See the appropriate reference documentation at spring.io/projects/spring-data.

Spring Boot also provides auto-configuration for the InfluxDB client but it is deprecated in favor of the new InfluxDB Java client that provides its own Spring Boot integration.

### 9.2.1. Redis

Redis is a cache, message broker, and richly-featured key-value store. Spring Boot offers basic auto-configuration for the Lettuce and Jedis client libraries and the abstractions on top of them provided by Spring Data Redis.

There is a `spring-boot-starter-data-redis` "Starter" for collecting the dependencies in a convenient way. By default, it uses Lettuce. That starter handles both traditional and reactive applications.

| TIP | We also provide a `spring-boot-starter-data-redis-reactive` "Starter" for consistency with the other stores with reactive support. |
|---|---|

#### Connecting to Redis

You can inject an auto-configured `RedisConnectionFactory`, `StringRedisTemplate`, or vanilla `RedisTemplate` instance as you would any other Spring Bean. The following listing shows an example of such a bean:

*Java*

```
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final StringRedisTemplate template;

    public MyBean(StringRedisTemplate template) {
        this.template = template;
    }

    public Boolean someMethod() {
        return this.template.hasKey("spring");
    }

}
```

*Kotlin*

```
import org.springframework.data.redis.core.StringRedisTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: StringRedisTemplate) {

    fun someMethod(): Boolean {
        return template.hasKey("spring")
    }

}
```

By default, the instance tries to connect to a Redis server at `localhost:6379`. You can specify custom connection details using `spring.data.redis.*` properties, as shown in the following example:

*Properties*

```
spring.data.redis.host=localhost
spring.data.redis.port=6379
spring.data.redis.database=0
spring.data.redis.username=user
spring.data.redis.password=secret
```

*Yaml*

```
spring:
  data:
    redis:
      host: "localhost"
      port: 6379
      database: 0
      username: "user"
      password: "secret"
```

**TIP**

You can also register an arbitrary number of beans that implement `LettuceClientConfigurationBuilderCustomizer` for more advanced customizations. `ClientResources` can also be customized using `ClientResourcesBuilderCustomizer`. If you use Jedis, `JedisClientConfigurationBuilderCustomizer` is also available. Alternatively, you can register a bean of type `RedisStandaloneConfiguration`, `RedisSentinelConfiguration`, or `RedisClusterConfiguration` to take full control over the configuration.

If you add your own `@Bean` of any of the auto-configured types, it replaces the default (except in the case of `RedisTemplate`, when the exclusion is based on the bean name, `redisTemplate`, not its type).

By default, a pooled connection factory is auto-configured if `commons-pool2` is on the classpath.

The auto-configured `RedisConnectionFactory` can be configured to use SSL for communication with the server by setting the properties as shown in this example:

*Properties*

```
spring.data.redis.ssl.enabled=true
```

*Yaml*

```
spring:
  data:
    redis:
      ssl:
        enabled: true
```

Custom SSL trust material can be configured in an SSL bundle and applied to the `RedisConnectionFactory` as shown in this example:

*Properties*

```
spring.data.redis.ssl.bundle=example
```

*Yaml*

```
spring:
  data:
    redis:
      ssl:
        bundle: "example"
```

### 9.2.2. MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the `spring-boot-starter-data-mongodb` and `spring-boot-starter-data-mongodb-reactive` "Starters".

#### Connecting to a MongoDB Database

To access MongoDB databases, you can inject an auto-configured `org.springframework.data.mongodb.MongoDatabaseFactory`. By default, the instance tries to connect to a MongoDB server at `mongodb://localhost/test`. The following example shows how to connect to a MongoDB database:

*Java*

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;

import org.springframework.data.mongodb.MongoDatabaseFactory;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final MongoDatabaseFactory mongo;

    public MyBean(MongoDatabaseFactory mongo) {
        this.mongo = mongo;
    }

    public MongoCollection<Document> someMethod() {
        MongoDatabase db = this.mongo.getMongoDatabase();
        return db.getCollection("users");
    }

}
```

*Kotlin*

```
import com.mongodb.client.MongoCollection
import org.bson.Document
import org.springframework.data.mongodb.MongoDatabaseFactory
import org.springframework.stereotype.Component

@Component
class MyBean(private val mongo: MongoDatabaseFactory) {

    fun someMethod(): MongoCollection<Document> {
        val db = mongo.mongoDatabase
        return db.getCollection("users")
    }

}
```

If you have defined your own `MongoClient`, it will be used to auto-configure a suitable `MongoDatabaseFactory`.

The auto-configured `MongoClient` is created using a `MongoClientSettings` bean. If you have defined your own `MongoClientSettings`, it will be used without modification and the `spring.data.mongodb` properties will be ignored. Otherwise a `MongoClientSettings` will be auto-configured and will have the `spring.data.mongodb` properties applied to it. In either case, you can declare one or more

`MongoClientSettingsBuilderCustomizer` beans to fine-tune the `MongoClientSettings` configuration. Each will be called in order with the `MongoClientSettings.Builder` that is used to build the `MongoClientSettings`.

You can set the `spring.data.mongodb.uri` property to change the URL and configure additional settings such as the *replica set*, as shown in the following example:

*Properties*

```
spring.data.mongodb.uri=mongodb://user:secret@mongoserver1.example.com:27017,mongoserv
er2.example.com:23456/test
```

*Yaml*

```
spring:
  data:
    mongodb:
      uri:
"mongodb://user:secret@mongoserver1.example.com:27017,mongoserver2.example.com:23456/t
est"
```

Alternatively, you can specify connection details using discrete properties. For example, you might declare the following settings in your `application.properties`:

*Properties*

```
spring.data.mongodb.host=mongoserver1.example.com
spring.data.mongodb.port=27017
spring.data.mongodb.additional-hosts[0]=mongoserver2.example.com:23456
spring.data.mongodb.database=test
spring.data.mongodb.username=user
spring.data.mongodb.password=secret
```

*Yaml*

```
spring:
  data:
    mongodb:
      host: "mongoserver1.example.com"
      port: 27017
      additional-hosts:
      - "mongoserver2.example.com:23456"
      database: "test"
      username: "user"
      password: "secret"
```

The auto-configured `MongoClient` can be configured to use SSL for communication with the server by setting the properties as shown in this example:

*Properties*

```
spring.data.mongodb.uri=mongodb://user:secret@mongoserver1.example.com:27017,mongoserv
er2.example.com:23456/test
spring.data.mongodb.ssl.enabled=true
```

*Yaml*

```
spring:
  data:
    mongodb:
      uri:
"mongodb://user:secret@mongoserver1.example.com:27017,mongoserver2.example.com:23456/t
est"
      ssl:
        enabled: true
```

Custom SSL trust material can be configured in an SSL bundle and applied to the `MongoClient` as shown in this example:

*Properties*

```
spring.data.mongodb.uri=mongodb://user:secret@mongoserver1.example.com:27017,mongoserv
er2.example.com:23456/test
spring.data.mongodb.ssl.bundle=example
```

*Yaml*

```
spring:
  data:
    mongodb:
      uri:
"mongodb://user:secret@mongoserver1.example.com:27017,mongoserver2.example.com:23456/t
est"
      ssl:
        bundle: "example"
```

**TIP**

If `spring.data.mongodb.port` is not specified, the default of `27017` is used. You could delete this line from the example shown earlier.

You can also specify the port as part of the host address by using the `host:port` syntax. This format should be used if you need to change the port of an `additional-hosts` entry.

| | |
|---|---|
| **TIP** | If you do not use Spring Data MongoDB, you can inject a `MongoClient` bean instead of using `MongoDatabaseFactory`. If you want to take complete control of establishing the MongoDB connection, you can also declare your own `MongoDatabaseFactory` or `MongoClient` bean. |
| **NOTE** | If you are using the reactive driver, Netty is required for SSL. The auto-configuration configures this factory automatically if Netty is available and the factory to use has not been customized already. |

#### MongoTemplate

Spring Data MongoDB provides a `MongoTemplate` class that is very similar in its design to Spring's `JdbcTemplate`. As with `JdbcTemplate`, Spring Boot auto-configures a bean for you to inject the template, as follows:

*Java*

```
import com.mongodb.client.MongoCollection;
import org.bson.Document;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final MongoTemplate mongoTemplate;

    public MyBean(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public MongoCollection<Document> someMethod() {
        return this.mongoTemplate.getCollection("users");
    }

}
```

*Kotlin*

```
import com.mongodb.client.MongoCollection
import org.bson.Document
import org.springframework.data.mongodb.core.MongoTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val mongoTemplate: MongoTemplate) {

    fun someMethod(): MongoCollection<Document> {
        return mongoTemplate.getCollection("users")
    }

}
```

See the `MongoOperations` Javadoc for complete details.

### Spring Data MongoDB Repositories

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed automatically, based on method names.

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a MongoDB data class rather than a JPA `@Entity`, it works in the same way, as shown in the following example:

*Java*

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

public interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    City findByNameAndStateAllIgnoringCase(String name, String state);

}
```

*Kotlin*

```
import org.springframework.data.domain.Page
import org.springframework.data.domain.Pageable
import org.springframework.data.repository.Repository

interface CityRepository :
    Repository<City?, Long?> {
    fun findAll(pageable: Pageable?): Page<City?>?
    fun findByNameAndStateAllIgnoringCase(name: String?, state: String?): City?
}
```

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by using `@EnableMongoRepositories` and `@EntityScan` respectively.

**TIP** For complete details of Spring Data MongoDB, including its rich object mapping technologies, see its reference documentation.

### 9.2.3. Neo4j

Neo4j is an open-source NoSQL graph database that uses a rich data model of nodes connected by first class relationships, which is better suited for connected big data than traditional RDBMS approaches. Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` "Starter".

#### Connecting to a Neo4j Database

To access a Neo4j server, you can inject an auto-configured `org.neo4j.driver.Driver`. By default, the instance tries to connect to a Neo4j server at `localhost:7687` using the Bolt protocol. The following example shows how to inject a Neo4j `Driver` that gives you access, amongst other things, to a `Session`:

*Java*

```
import org.neo4j.driver.Driver;
import org.neo4j.driver.Session;
import org.neo4j.driver.Values;

import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final Driver driver;

    public MyBean(Driver driver) {
        this.driver = driver;
    }

    public String someMethod(String message) {
        try (Session session = this.driver.session()) {
            return session.executeWrite(
                    (transaction) -> transaction
                        .run("CREATE (a:Greeting) SET a.message = $message RETURN
a.message + ', from node ' + id(a)",
                                Values.parameters("message", message))
                        .single()
                        .get(0)
                        .asString());
        }
    }

}
```

*Kotlin*

```
import org.neo4j.driver.*
import org.springframework.stereotype.Component

@Component
class MyBean(private val driver: Driver) {
    fun someMethod(message: String?): String {
        driver.session().use { session ->
            return@someMethod session.executeWrite { transaction: TransactionContext
->
                transaction
                    .run(
                        "CREATE (a:Greeting) SET a.message = \$message RETURN
a.message + ', from node ' + id(a)",
                        Values.parameters("message", message)
                    )
                    .single()[0].asString()
            }
        }
    }
}
```

You can configure various aspects of the driver using `spring.neo4j.*` properties. The following example shows how to configure the uri and credentials to use:

*Properties*

```
spring.neo4j.uri=bolt://my-server:7687
spring.neo4j.authentication.username=neo4j
spring.neo4j.authentication.password=secret
```

*Yaml*

```
spring:
  neo4j:
    uri: "bolt://my-server:7687"
    authentication:
      username: "neo4j"
      password: "secret"
```

The auto-configured `Driver` is created using `ConfigBuilder`. To fine-tune its configuration, declare one or more `ConfigBuilderCustomizer` beans. Each will be called in order with the `ConfigBuilder` that is used to build the `Driver`.

#### Spring Data Neo4j Repositories

Spring Data includes repository support for Neo4j. For complete details of Spring Data Neo4j, see the reference documentation.

Spring Data Neo4j shares the common infrastructure with Spring Data JPA as many other Spring Data modules do. You could take the JPA example from earlier and define `City` as Spring Data Neo4j `@Node` rather than JPA `@Entity` and the repository abstraction works in the same way, as shown in the following example:

*Java*

```
import java.util.Optional;

import org.springframework.data.neo4j.repository.Neo4jRepository;

public interface CityRepository extends Neo4jRepository<City, Long> {

    Optional<City> findOneByNameAndState(String name, String state);

}
```

*Kotlin*

```
import org.springframework.data.neo4j.repository.Neo4jRepository
import java.util.Optional

interface CityRepository : Neo4jRepository<City?, Long?> {

    fun findOneByNameAndState(name: String?, state: String?): Optional<City?>?

}
```

The `spring-boot-starter-data-neo4j` "Starter" enables the repository support as well as transaction management. Spring Boot supports both classic and reactive Neo4j repositories, using the `Neo4jTemplate` or `ReactiveNeo4jTemplate` beans. When Project Reactor is available on the classpath, the reactive style is also auto-configured.

Repositories and entities are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and entities by using `@EnableNeo4jRepositories` and `@EntityScan` respectively.

**NOTE**

In an application using the reactive style, a `ReactiveTransactionManager` is not auto-configured. To enable transaction management, the following bean must be defined in your configuration:

*Java*

```
import org.neo4j.driver.Driver;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.data.neo4j.core.ReactiveDatabaseSelectionProvider;
import
org.springframework.data.neo4j.core.transaction.ReactiveNeo4jTransaction
Manager;

@Configuration(proxyBeanMethods = false)
public class MyNeo4jConfiguration {

    @Bean
    public ReactiveNeo4jTransactionManager
reactiveTransactionManager(Driver driver,
            ReactiveDatabaseSelectionProvider databaseNameProvider) {
        return new ReactiveNeo4jTransactionManager(driver,
databaseNameProvider);
    }

}
```

*Kotlin*

```
import org.neo4j.driver.Driver
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import
org.springframework.data.neo4j.core.ReactiveDatabaseSelectionProvider
import
org.springframework.data.neo4j.core.transaction.ReactiveNeo4jTransaction
Manager

@Configuration(proxyBeanMethods = false)
class MyNeo4jConfiguration {

    @Bean
    fun reactiveTransactionManager(driver: Driver,
            databaseNameProvider: ReactiveDatabaseSelectionProvider):
ReactiveNeo4jTransactionManager {
        return ReactiveNeo4jTransactionManager(driver,
databaseNameProvider)
    }
}
```

### 9.2.4. Elasticsearch

Elasticsearch is an open source, distributed, RESTful search and analytics engine. Spring Boot offers basic auto-configuration for Elasticsearch clients.

Spring Boot supports several clients:

- The official low-level REST client
- The official Java API client
- The `ReactiveElasticsearchClient` provided by Spring Data Elasticsearch

Spring Boot provides a dedicated "Starter", `spring-boot-starter-data-elasticsearch`.

#### Connecting to Elasticsearch Using REST clients

Elasticsearch ships two different REST clients that you can use to query a cluster: the low-level client from the `org.elasticsearch.client:elasticsearch-rest-client` module and the Java API client from the `co.elastic.clients:elasticsearch-java` module. Additionally, Spring Boot provides support for a reactive client from the `org.springframework.data:spring-data-elasticsearch` module. By default, the clients will target `localhost:9200`. You can use `spring.elasticsearch.*` properties to further tune how the clients are configured, as shown in the following example:

*Properties*

```
spring.elasticsearch.uris=https://search.example.com:9200
spring.elasticsearch.socket-timeout=10s
spring.elasticsearch.username=user
spring.elasticsearch.password=secret
```

*Yaml*

```
spring:
  elasticsearch:
    uris: "https://search.example.com:9200"
    socket-timeout: "10s"
    username: "user"
    password: "secret"
```

##### Connecting to Elasticsearch Using RestClient

If you have `elasticsearch-rest-client` on the classpath, Spring Boot will auto-configure and register a `RestClient` bean. In addition to the properties described previously, to fine-tune the `RestClient` you can register an arbitrary number of beans that implement `RestClientBuilderCustomizer` for more advanced customizations. To take full control over the clients' configuration, define a `RestClientBuilder` bean.

Additionally, if `elasticsearch-rest-client-sniffer` is on the classpath, a `Sniffer` is auto-configured to automatically discover nodes from a running Elasticsearch cluster and set them on the `RestClient` bean. You can further tune how `Sniffer` is configured, as shown in the following

example:

*Properties*

```
spring.elasticsearch.restclient.sniffer.interval=10m
spring.elasticsearch.restclient.sniffer.delay-after-failure=30s
```

*Yaml*

```
spring:
  elasticsearch:
    restclient:
      sniffer:
        interval: "10m"
        delay-after-failure: "30s"
```

#### Connecting to Elasticsearch Using ElasticsearchClient

If you have `co.elastic.clients:elasticsearch-java` on the classpath, Spring Boot will auto-configure and register an `ElasticsearchClient` bean.

The `ElasticsearchClient` uses a transport that depends upon the previously described `RestClient`. Therefore, the properties described previously can be used to configure the `ElasticsearchClient`. Furthermore, you can define a `RestClientOptions` bean to take further control of the behavior of the transport.

#### Connecting to Elasticsearch using ReactiveElasticsearchClient

Spring Data Elasticsearch ships `ReactiveElasticsearchClient` for querying Elasticsearch instances in a reactive fashion. If you have Spring Data Elasticsearch and Reactor on the classpath, Spring Boot will auto-configure and register a `ReactiveElasticsearchClient`.

The `ReactiveElasticsearchclient` uses a transport that depends upon the previously described `RestClient`. Therefore, the properties described previously can be used to configure the `ReactiveElasticsearchClient`. Furthermore, you can define a `RestClientOptions` bean to take further control of the behavior of the transport.

### Connecting to Elasticsearch by Using Spring Data

To connect to Elasticsearch, an `ElasticsearchClient` bean must be defined, auto-configured by Spring Boot or manually provided by the application (see previous sections). With this configuration in place, an `ElasticsearchTemplate` can be injected like any other Spring bean, as shown in the following example:

*Java*

```
import org.springframework.data.elasticsearch.client.elc.ElasticsearchTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final ElasticsearchTemplate template;

    public MyBean(ElasticsearchTemplate template) {
        this.template = template;
    }

    public boolean someMethod(String id) {
        return this.template.exists(id, User.class);
    }

}
```

*Kotlin*

```
import org.springframework.stereotype.Component

@Component
class MyBean(private val template:
org.springframework.data.elasticsearch.client.elc.ElasticsearchTemplate ) {

    fun someMethod(id: String): Boolean {
        return template.exists(id, User::class.java)
    }

}
```

In the presence of `spring-data-elasticsearch` and Reactor, Spring Boot can also auto-configure a ReactiveElasticsearchClient and a `ReactiveElasticsearchTemplate` as beans. They are the reactive equivalent of the other REST clients.

#### Spring Data Elasticsearch Repositories

Spring Data includes repository support for Elasticsearch. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now an Elasticsearch `@Document` class rather than a JPA `@Entity`, it works in the same way.

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by