### Configuring `DataSource` in Spring Boot

In Spring Boot, configuring a `DataSource` (which is responsible for providing database connections) is straightforward. Spring Boot provides auto-configuration for `DataSource` out of the box, and it automatically configures a connection pool such as \*\*HikariCP\*\*, \*\*Tomcat\*\*, or \*\*Commons DBCP\*\* when the required dependencies are present. However, you can also manually configure the `DataSource` to customize how database connections are handled.

### 1. \*\*Using Auto-Configuration (Default DataSource)\*\*

Spring Boot automatically configures the `DataSource` based on the database properties defined in `application.properties` or `application.yml`. Spring Boot uses \*\*HikariCP\*\* as the default connection pool (from Spring Boot 2.x onwards).

#### Example `application.properties`:

```properties

# DataSource Configuration

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.initialization-mode=always # Initialize schema on startup

```

- `spring.datasource.url`: The JDBC URL for connecting to the database.

- `spring.datasource.username`: The username used to connect to the database.

- `spring.datasource.password`: The password for the database connection.

- `spring.datasource.driver-class-name`: The class name of the JDBC driver (optional, Spring Boot auto-detects it based on the URL).

Spring Boot automatically sets up a `DataSource` bean with this configuration, so there is no need to write any custom configuration code for the `DataSource`.

---

### 2. \*\*Using `application.yml` for Configuration\*\*

You can also configure the `DataSource` in `application.yml` instead of `application.properties`.

```yaml

spring:

datasource:

url: jdbc:mysql://localhost:3306/mydb

username: root

password: root

driver-class-name: com.mysql.cj.jdbc.Driver

initialization-mode: always

```

This configuration achieves the same as the `application.properties` file but is in YAML format.

---

### 3. \*\*Customizing the Connection Pool (HikariCP)\*\*

Spring Boot defaults to \*\*HikariCP\*\* for database connection pooling, but you can customize HikariCP settings as needed.

\*\*Example `application.properties`:\*\*

```properties

# HikariCP Settings

spring.datasource.hikari.maximum-pool-size=10

spring.datasource.hikari.minimum-idle=5

spring.datasource.hikari.idle-timeout=30000

spring.datasource.hikari.connection-timeout=20000

spring.datasource.hikari.max-lifetime=1800000

```

- `spring.datasource.hikari.maximum-pool-size`: Maximum number of connections in the pool.

- `spring.datasource.hikari.minimum-idle`: Minimum number of idle connections maintained in the pool.

- `spring.datasource.hikari.idle-timeout`: Maximum amount of time a connection can remain idle before being removed.

- `spring.datasource.hikari.connection-timeout`: Maximum time to wait for a connection from the pool.

- `spring.datasource.hikari.max-lifetime`: Maximum lifetime of a connection in the pool.

---

### 4. \*\*Manual DataSource Configuration\*\*

If you need more control over how the `DataSource` is created (for example, if you're using multiple databases or want to use a specific connection pool), you can manually configure the `DataSource` bean in your Spring Boot application.

#### Example `DataSource` Bean Configuration:

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import javax.sql.DataSource;

@Configuration

public class DataSourceConfig {

@Bean

@ConfigurationProperties(prefix = "spring.datasource")

public DataSource dataSource() {

return DataSourceBuilder.create().build();

}

}

```

In this example, the `DataSource` bean is manually configured by reading properties prefixed with `spring.datasource` from the configuration files. The `DataSourceBuilder` helps create a `DataSource` with the properties provided.

---

### 5. \*\*Multiple DataSources (Primary and Secondary)\*\*

If your application requires multiple `DataSource` instances (for example, to connect to different databases), you can configure multiple `DataSource` beans in Spring Boot.

#### Example:

\*\*`application.properties`\*\*:

```properties

# Primary DataSource

spring.datasource.primary.url=jdbc:mysql://localhost:3306/primarydb

spring.datasource.primary.username=root

spring.datasource.primary.password=root

# Secondary DataSource

spring.datasource.secondary.url=jdbc:mysql://localhost:3306/secondarydb

spring.datasource.secondary.username=root

spring.datasource.secondary.password=root

```

\*\*Configuration Class:\*\*

```java

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.datasource.lookup.JndiDataSourceLookup;

import javax.sql.DataSource;

@Configuration

public class MultiDataSourceConfig {

@Bean(name = "primaryDataSource")

@ConfigurationProperties(prefix = "spring.datasource.primary")

public DataSource primaryDataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "secondaryDataSource")

@ConfigurationProperties(prefix = "spring.datasource.secondary")

public DataSource secondaryDataSource() {

return DataSourceBuilder.create().build();

}

}

```

Here, we configure two `DataSource` beans, one for the primary database and one for the secondary database. We can then use `@Qualifier` to specify which `DataSource` to inject when needed.

---

### 6. \*\*Using JNDI DataSource (Java Naming and Directory Interface)\*\*

In enterprise applications, the `DataSource` is often provided by the application server via JNDI. Spring Boot supports this configuration as well.

#### Example:

\*\*`application.properties`\*\*:

```properties

spring.datasource.jndi-name=java:/comp/env/jdbc/myDataSource

```

\*\*Configuration Class\*\*:

```java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.datasource.lookup.JndiDataSourceLookup;

import javax.sql.DataSource;

@Configuration

public class JndiDataSourceConfig {

@Bean

public DataSource dataSource() {

JndiDataSourceLookup dataSourceLookup = new JndiDataSourceLookup();

return dataSourceLookup.getDataSource("java:/comp/env/jdbc/myDataSource");

}

}

```

In this example, the `DataSource` is retrieved from the JNDI registry, which is often configured by the application server (such as Tomcat or WebLogic).

---

### Conclusion

Spring Boot simplifies `DataSource` configuration by providing out-of-the-box support through auto-configuration. With default settings (such as HikariCP connection pooling), it's easy to get started by simply defining database properties. However, when needed, Spring Boot allows manual configuration for more advanced use cases such as custom connection pools, multiple databases, or JNDI lookups.