Connection Pooling with Spring BOOT

**Connection Pooling?**

JDBC API is used for establishing connection with JDBC in middle-tier server environment. Creating connection is resource-expensive and leads to performance degrade in high scale applications. In this type of environment, performance can be improved significantly when connection pooling is used. Connection pooling means that connections are reused rather than created each time a connection is requested. To facilitate connection reuse, a memory cache of database connections, called a connection pool, is maintained by a connection pooling module as a layer on top of any standard JDBC driver product.

Connection pooling is performed in the background and does not affect how an application is coded; however, the application must use a DataSource object.

**Connection Pooling in Spring?**

Setting up JDBC Database Connection Pool in Spring framework is easy for any Java application by adding configuration in spring configuration file. In case you are using spring boot, this becomes more easier to add connection pooling configuration.

All we need to do is to set up data source properties for pooling apart from connection properties. Usually, while configuring any spring application (using xml or java bean based), we define a DataSource bean with connection properties as *driver class name, jdbc url, username, password* etc. This make DataSource object to provide connection based on given properties.

If using core JDBC api to connect to db then from same datasource, we get the connection as *dataSource.getConnection()*. If using spring-jdbc then we create JdbcTemplate by passing this datasource.

Most used connection pooling frameworks are: DBCP2, tomcat, hikariCP, c3p0, vibur.

Among these, hikariCP is considered as better in terms of performance with below stats:

A screenshot of a cell phone

Description generated with very high confidence

* One Connection Cycle is defined as single DataSource.getConnection()/Connection.close().
* One Statement Cycle is defined as single Connection.prepareStatement(), Statement.execute(), Statement.close().

However, these may effect at very large scale of jdbc operation.

Read more: <https://github.com/brettwooldridge/HikariCP#jmh-benchmarks-checkered_flag>

These are following component to provide basic idea of Spring Batch.

Click with ctrl to navigate to these topics.

1. [Using DBCP2 for connection pooling](#dpcp1)

1. [HikariCP for connection pooling](#hikaricp2)
2. [Connection Pooling in Spring Boot](#poolinboot3)
3. [Connection to a JNDI DataSource](#jndi4)
4. **Using DBCP2 for connection pooling**

In application, interaction with database is common and often take milliseconds to execute its operation. But, creating a separate connection for each request per user will more time than executing the operation. The commons-dbcp2 package relies on code in the commons-pool2 package to provide the underlying object pool mechanisms that it utilizes.

DBCP now comes in three different versions to support different versions of JDBC. Here is how it works:

* DBCP 2 compiles and runs under Java 7 only (JDBC 4.1)
* DBCP 1.4 compiles and runs under Java 6 only (JDBC 4)
* DBCP 1.3 compiles and runs under Java 1.4-5 only (JDBC 3)

To use dbcp2 as connection pooling framework in spring requires following dependency:

compile group: 'org.apache.commons', name: 'commons-dbcp2', version: '2.1.1'

this will make available BasicDataSource to spring application for further configuration.

**DataSource Configuration:**

Apart from common configurations, we can many properties related to connection pooling as:

Common Properties:

username The connection username to be passed to our JDBC driver to establish a connection.

password The connection password to be passed to our JDBC driver to establish a connection.

url The connection URL to be passed to our JDBC driver to establish a connection.

driverClassName The fully qualified Java class name of the JDBC driver to be used.

Pool properties:

These below properties specifies properties for connection in pool along with their life time.

|  |  |  |
| --- | --- | --- |
| Property | Default | Description |
| initialSize | 0 | The initial number of connections that are created when the pool is started. |
| maxTotal | 8 | The maximum number of active connections that can be allocated from this pool at the same time, or negative for no limit. |
| maxIdle | 8 | The maximum number of connections that can remain idle in the pool, without extra ones being released, or negative for no limit. |
| minIdle | 0 | The minimum number of connections that can remain idle in the pool, without extra ones being created, or zero to create none. |
| maxWaitMillis | indefinitely | The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception, or -1 to wait indefinitely. |
| maxConnLifetimeMillis | -1 | The maximum lifetime in milliseconds of a connection. After this time is exceeded the connection will fail the next activation, passivation or validation test. A value of zero or less means the connection has an infinite lifetime. |

sample configuration for above properties:

@Bean

**public** BasicDataSource pooledDataSource() {

BasicDataSource dataSource = **new** BasicDataSource();

dataSource.setDriverClassName(driverClassName);

dataSource.setUrl(url);

dataSource.setUsername(username);

dataSource.setPassword(password);

//Initial Pool size after start

dataSource.setInitialSize(5);

//Max number of active connections [ default - 8]

dataSource.setMaxTotal(10);

//Max number of connections which may remain idle in pool [ default - 8]

dataSource.setMaxIdle(2);

//Max lifetime of a connection

dataSource.setMaxConnLifetimeMillis(15000);

//Max time pool will wait for returning connection, if no connection remains in pool

dataSource.setMaxWaitMillis(30000);

**return** dataSource;

}

Read More properties: <https://commons.apache.org/proper/commons-dbcp/configuration.html>

Now, this created datasource can be directly used in application or can be used by JdbcTemplate or others to perform database related operations as below:

@Bean

@Scope("prototype")

**public** JdbcTemplate jdbcTemplate() {

**return** **new** JdbcTemplate(pooledDataSource(), **true**);

}

1. **HikariCP for connection Pooling**

We have already seen diagram depicting HikariCP to be better. Let’s see how to configure in spring application. HikariCP comes with the support for all the main versions of JVM. Each version requires its dependency

Java 8: compile ('com.zaxxer:HikariCP:2.6.1')

Java 7: compile ('com.zaxxer:HikariCP-java7: 2.4.13')

Java 6: compile ('com.zaxxer:HikariCP-java6: 2.3.13')

These above dependencies of gradle will make HikariDataSource to be available for using in application. Now, let’s consider other properties configurations of HikariCP, common properties regrading connectivity will be same as dbcp2 or any other datasource.

HikariCP specific pool properties are:

|  |  |  |
| --- | --- | --- |
| Property | Default | Description |
| maximumPoolSize | 10 | The maximum size that the pool can reach, including both idle and in-use connections. Basically, this value will determine the maximum number of actual connections to the database backend. |
| maxLifetime | 1800000  (30 minutes) | The maximum lifetime of a connection in the pool. An in-use connection will never be retired, only when it is closed will it then be removed. Recommended to set this value, and it should be at least 30 seconds. |
| idleTimeout | 600000  (10 minutes) | The maximum amount of time that a connection can sit idle in the pool. This setting only applies when minimum-Idle is defined to be less than maximumPoolSize. |
| minimumIdle | as maximum  PoolSize | The minimum number of idle connections that HikariCP tries to maintain in the pool. |
| poolName | auto-generated | This property represents a user-defined name for the connection pool and appears mainly in logging consoles to identify pools and pool configurations. |
| connectionTimeout | 30000  (30 seconds) | The maximum number of milliseconds that a client (that's you) will wait for a connection from the pool. If this time is exceeded without a connection becoming available, a SQLException will be thrown. |

sample configuration for above properties:

@Bean

**public** HikariDataSource pooledDataSource() {

HikariDataSource dataSource = **new** HikariDataSource();

dataSource.setDriverClassName(driverClassName);

dataSource.setJdbcUrl(url);

dataSource.setUsername(username);

dataSource.setPassword(password);

dataSource.setMaximumPoolSize(10);

dataSource.setMinimumIdle(2);

dataSource.setMaxLifetime(30000);

dataSource.setIdleTimeout(10000);

**return** dataSource;

}

Read More properties: <https://commons.apache.org/proper/commons-dbcp/configuration.html>

1. **Connection pooling in Spring Boot**

As discussed in previous section, how to configure data source with connection pooling in spring application. Now let’s see how we can do the same in spring boot application. Previous ways of configuration will make simpler if we utilize spring boot autoconfiguration attribute.

Spring Boot dependency - **spring-boot-starter-jdbc** will get tomcat-jdbc as dependency. Spring Boot follow below auto-configuration for data source configuration. Spring Boot prefers tomcat-jdbc as default connection pooling for data source and follow below sequence to find connection:

**Tomcat pool -->> - HikariCP -->> Commons DBCP -->> Commons DBCP2**

Note: Commons DBCP is not recommended because it is deprecated.

Based on dependencies add, spring boot will make DataSource to be available to application with default properties. DataSource configuration is controlled by external configuration properties in *spring.datasource.\**. For example, you might declare the following section in application.properties:

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=dbuser

spring.datasource.password=dbpass

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

These above configurations will affect any datasource configuration mentioned above. Like in previous sections we discussed about pooled datasource properties of HikariCP & DBCP2, we can add those properties directly from same application.properties file using respective prefix:

spring.datasource.tomcat.\*, spring.datasource.hikari.\*, and spring.datasource.dbcp2.\*

To customize tomcat connection pool properties:

spring.datasource.tomcat.max-wait=10000

spring.datasource.tomcat.max-active=50

spring.datasource.tomcat.test-on-borrow=true

To customize commons DBCP2 pool properties:

spring.datasource.dbcp2.initial-size=5

spring.datasource.dbcp2.max-total=10

spring.datasource.dbcp2.max-idle=2

To customize hikariCP connection pool properties:

spring.datasource.hikari.connection-timeout=70000

spring.datasource.hikari.maximum-pool-size=6

spring.datasource.hikari.max-life-time=150000

spring.datasource.hikari.minimum-idle=3

spring.datasource.hikari.pool-name=zeta-hikari

spring.datasource.hikari.initialization-fail-timeout=0

Similarly, other properties can also be added. To use in application, add required dependency to application and autowire the datasource.

1. **Connection to a JNDI DataSource**

We can set up data source connection pooling using application server embedded connection pooling support. For example, spring boot starter project by default have embedded tomcat server. So, to use tomcat connection pooling facility, we need to configure it jndi resource and naming lookup.

It is like adding resource file to server.xml to any external tomcat server. To add these property, we can make use of it by defining bean of TomcatEmbeddedServletContainerFactory. Setting this up will enable naming lookup for jndi name and resource link. A complete configuration is:

@Bean

**public** TomcatEmbeddedServletContainerFactory tomcatFactory() {

**return** **new** TomcatEmbeddedServletContainerFactory() {

@Override

**protected** TomcatEmbeddedServletContainer getTomcatEmbeddedServletContainer(

Tomcat tomcat) {

tomcat.enableNaming();

**return** **super**.getTomcatEmbeddedServletContainer(tomcat);

}

@Override

**protected** **void** postProcessContext(Context context) {

ContextResource resource = **new** ContextResource();

resource.setName(jndiName);

resource.setType(DataSource.**class**.getName());

resource.setProperty("driverClassName", driverClassName);

resource.setProperty("url", url);

resource.setProperty("username", "devuser");

resource.setProperty("password", "leo$123");

resource.setProperty("maxActive", "10"); connection pool properties

resource.setProperty("maxIdle", "5");

resource.setProperty("minIdle", "2");

resource.setProperty("maxWait", "10000");

context.getNamingResources().addResource(resource);

}

};

}

@Bean(destroyMethod = "")

**public** DataSource jndiDataSource() **throws** IllegalArgumentException, NamingException {

JndiObjectFactoryBean bean = **new** JndiObjectFactoryBean();

bean.setJndiName("java:comp/env/jdbc/dataSource");

bean.setProxyInterface(DataSource.**class**);

bean.setLookupOnStartup(**false**);

bean.afterPropertiesSet();

**return** (DataSource) bean.getObject();

}

The above configuration will add connection properties and tomcat connection properties to running embedded tomcat server to make use of jndi data source. The second will create DataSource by looking up the resource that you just registered using the Jndi name, which will be used to inject DataSource into the application. Other connection pooling properties can be also added in same resource object as few are already added for more properties details:

Read-more: <https://tomcat.apache.org/tomcat-8.0-doc/jdbc-pool.html>