Blog Zenika

Using Tomcat JDBC connection pool in a standalone environment

3

PAR ARNAUD COGOLUÈGNES LE 30 JANVIER 2013

JAVA

A multi-user application working against a database cannot be efficient if it doesn't use connection pooling. Middleware can offer this service, but not all applications rely on such middleware. These applications must then come up with their own way to pool connections. The Apache Tomcat project doesn't only come with the most popular web container but also with a performant connection pool library, Tomcat JDBC. This post covers how to configure Tomcat JDBC in a typical Maven + Spring application.

Why using a standalone connection pool?

Here are some good reasons to use a standalone connection pool:

- the application is running in a container, but you don't want to use the container connection pool (not efficient or proprietary configuration mechanism like a web console or some complex XML files, etc). The application then creates the connection when it starts. The connection parameters (driver, url, user, password) and pool configuration can be externalized in a simple properties, text file.
- the application needs some connection pools at runtime. This isn't a common requirement, but some systems like Business Intelligence applications need to connect to different databases and the connections are configured at runtime, by advanced users.
- the application isn't running in a container and needs to connect to a database. Container-less deployment is getting more and more popular, and, as an example, the Dropwizard micro-services framework uses Tomcat JDBC to manage its connection pool.

If your application falls in any of this use cases, it's a good candidate to use Tomcat JDBC. Tomcat JDBC was introduced in Tomcat 7, as a replacement to Commons DBCP (see some reasons here).

A couple of posts from the Tomcat Expert blog explains thoroughly the features of Tomcat JDBC and another post even provides a comparison between Tomcat JDBC, Commons DBCP, and C3P0. To make it short, Tomcat JDBC is simpler and faster than the other implementations, without sacrifying the features.

What the Tomcat Export posts miss is the use of the pool in a standalone environment. So this articles focuses on the use of Tomcat JDBC in a typical standalone environment (Maven, Spring), rather than on the features or the performances.

Adding the Maven dependencies

Tomcat JDBC is available on the public Maven repositories. This means you can easily grab the dependencies with your favorite build/dependency management tool (Maven, Gradle, Ivy). Here is the dependency code for Maven:

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jdbc</artifactId>
  <version>7.0.35</version>
</dependency>
```

If you deploy on Tomcat 7 or provide Tomcat JDBC as an infrastructure library, you can set the scope to 'provided':

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jdbc</artifactId>
  <version>7.0.35</version>
  <scope>provided</scope>
</dependency>
```

This way, Tomcat JDBC won't be included in the final archive.

Declaring a pool

Tomcat JDBC is straightforward to use: one needs to create an instance of `org.apache.tomcat.jdbc.pool.DataSource` and use the appropriate setters to configure the pool:

```
import org.apache.tomcat.jdbc.pool.DataSource;

(...)

DataSource ds = new DataSource();
ds.setDriverClassName("org.h2.Driver");
ds.setUrl("jdbc:h2:java-config");
ds.setUsername("sa");
ds.setPassword("");
ds.setInitialSize(5);
ds.setMaxActive(10);
ds.setMaxIdle(5);
ds.setMinIdle(2);
```

Note `org.apache.tomcat.jdbc.pool.DataSource` implements the `javax.sql.DataSource`, so an instance of the pool can be used anywhere you need a standard `DataSource`.

Using Tomcat JDBC with Spring (Java configuration)

Spring is a common choice in Java enterprise applications. The framework brings portability to applications, so it makes sense to use a standalone connection pool in a Spring application. Here is an example of using Tomcat JDBC with Spring Java-based configuration:

```
@Configuration
public class DataAccessConfiguration {

    @Bean(destroyMethod = "close")
    public javax.sql.DataSource datasource() {
        org.apache.tomcat.jdbc.pool.DataSource ds = new org.apache.tomcat.jdbc.pool.DataSource();
        ds.setDriverClassName("org.h2.Driver");
        ds.setUrl("jdbc:h2:java-config");
        ds.setUsername("sa");
        ds.setPassword("");
        ds.setInitialSize(5);
        ds.setMaxActive(10);
        ds.setMaxIdle(5);
        ds.setMinIdle(2);
        return ds;
    }

    @Bean public JdbcOperations tpl() {
        return new JdbcTemplate(datasource());
    }

}
```

Note the use of the `destroyMethod` attribute: Spring will call the `close` method when its container shuts down, to make the pool give the connections back to the database. Note also the declaration of `JdbcTemplate`: it needs a `javax.sql.DataSource` to be created and thus accepts Tomcat JDBC `DataSource` implementation.

Using Tomcat JDBC with Spring (XML configuration)

The XML configuration is still popular in Spring applications, especially for infrastructure components like a connection pool:

```
<bean id="dataSource" class="org.apache.tomcat.jdbc.pool.DataSource" destroy-method="close">
    <property name="driverClassName" value="org.h2.Driver" />
    <property name="url" value="jdbc:h2:mem:xml-config" />
    <property name="username" value="sa" />
    <property name="password" value="" />
```

```
    <property name="initialSize" value="5" />
    <property name="maxActive" value="10" />
    <property name="maxIdle" value="5" />
    <property name="minIdle" value="2" />
</bean>

<bean class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>
```

Note the use of the `destroy-method` attribute, to ensure the pool is closed when the Spring container shuts down.

NB: the configuration parameters doesn't have to be hard-coded! Spring provides several ways to externalize such parameters (property placeholders, ${...} like syntax, and the `Environment` abstraction with `PropertySource`). Take a look at the Spring documentation for more information.

Handy features: connection initialization SQL and validation

Tomcat JDBC provides many features. You will probably need 2 of them if you go a little bit further than the basic usage of the connection pool.

The first feature is the execution of some SQL instruction when *a new connection is created*. The SQL code instruction is thus executed only once for each connection. This comes in handy when connections need to be « tagged » by the application to make monitoring easier. For PostgreSQL, this can be done this way:

```
ds.setInitSQL("SET application_name = 'my-app'");
```

By doing this, the connections created by our pool instance will show up with the `my-app` value for the `application_name` column when executing `select * from pg_stat_activity`. Very useful for monitoring!

The other feature is connection validation. Some databases close open connections quite aggressively if they detect they're not used or a connection can be lost because of a network glitch. The pool can then execute a validation query every time the application borrows a connection. If the validation query fails, the pool assumes it's dead and creates a new one. This is all transparent for the application, which doesn't have to worry about dead or invalid connections.

The validation query has a setter in Tomcat JDBC:

```
1 ds.setValidationQuery("select 1");
```

Conclusion

Tomcat JDBC is a robust, lighweight, and performant connection pool library. It's a viable alternative to the older yet popular Commons DBCP project and it can be easily embedded in any application. Don't wait to give it a try!

Source code

PARTAGEZ CET ARTICLE.

A PROPOS DE L'AUTEUR

ARNAUD COGOLUÈGNES

SUR LE MÊME THÈME

22 MAI 2017 0

Some notes about Jax-RS, HTTP statuses and Tomcat

25 NOVEMBRE 2016 1

Spring MVC test dans un contexte sécurisé

18 NOVEMBRE 2016 0

Implémenter un mécanisme d'AOP, sans framework !

3 COMMENTAIRES

TAKIDOSO on 21 AOÛT 2014 15 H 22 MIN

Great article, but there is a question:
Tomcat provides its connection pool with JNDI. If I like to provide Tomcat-JDBC to a « sub-application » via JNDI, is there a a way to reuse the JNDI-Provider implemented within Tomcat, or am I forced to produce my own?
If Tomcat's JNDI-Provider can be reused, How is it to be done?

REPLY ›

MARC JOHNEN on 19 JANVIER 2016 19 H 35 MIN

Hi,
I cloned the source code and get an exception when running SpringJavaConfigTest.
Any idea.
Greetings Marc

java.lang.IllegalStateException: Failed to load ApplicationContext

REPLY >

KÅRE on 25 SEPTEMBRE 2017 20 H 14 MIN

I do not understand why there is no release method.

REPLY >

AJOUTER UN COMMENTAIRE

Votre commentaire

Votre nom

Votre adresse mail

Votre site

ENVOYER

Recherche... Rechercher

ARTICLES RÉCENTS

Metasploit : plongeon dans le framework

L'échange de données front-back efficace avec GraphQL

TensorFlow, le nouveau framework pour démocratiser le Deep Learning (1/3)

Retour sur la Dockercon Europe 2017

Mon passage à la ScalaIO 2017

COMMENTAIRES RÉCENTS

kamal dans Npm vs Yarn

Soufiane dans Premiers pas avec ElasticSearch (Partie 1)

Kåre dans Using Tomcat JDBC connection pool in a standalone environment

greg dans Retour sur La Nuit du Hack 2017

Moughaoui dans Premiers pas avec ElasticSearch (Partie 1)