Penguin Dreams

- **PenguinDreams**

  News
  Assignments
  Resume
  Tutorials
  ⊞ Projects
  Thesis
  ⊞ Scripts
  Web Portfolio
  Work Portfolio
  Computers

- **Archives**

  ⊞ October 2013 (2)
  ⊞ August 2013 (1)
  ⊞ May 2013 (1)
  ⊞ April 2013 (1)
  ⊞ March 2013 (2)
  ⊞ February 2013 (2)
  ⊞ October 2012 (1)
  ⊞ August 2012 (2)
  ⊞ January 2012 (1)
  ⊞ July 2011 (1)
  ⊞ May 2011 (3)
  ⊞ April 2011 (1)
  ⊞ June 2010 (1)
  ⊟ May 2010 (1)
  Running Beans Locally that use Application Server Data Sources
  ⊞ April 2010 (2)
  ⊞ March 2010 (1)
  ⊞ November 2009 (2)
  ⊞ April 2009 (3)
  ⊞ February 2009 (1)
  ⊞ January 2009 (1)
  ⊞ December 2008 (4)
  ⊞ November 2008 (3)
  ⊞ August 2008 (1)
  ⊞ July 2008 (1)
  ⊞ June 2008 (2)
  ⊞ April 2008 (1)
  ⊞ February 2008 (1)
  ⊞ January 2008 (1)
  ⊞ December 2007 (3)
  ⊞ October 2007 (1)
  ⊞ September 2007 (1)
  ⊞ August 2007 (2)
  ⊞ July 2007 (1)
  ⊞ June 2007 (2)
  ⊞ April 2007 (2)

- Search

« Building Java EAR files using Ant
My Account's Been Hacked (No It Hasn't) »

Running Beans Locally that use Application Server Data Sources

When writing J2EE web applications, web services, enterprise Java beans (EJBs) or other pieces of code that run on a Java application server such as RedHat's JBoss, IBM WebSphere or Apache Tomcat, a developer typically doesn't load database drivers or connect to the database directly. Instead, a context lookup must be made in order to get a `DataSource`, and from there a `Connection`. However what if one needs to run existing code locally, outside of the web server? This guide shows developers how to setup a local context so application server code can be run in a stand-alone application without modification.

In a typical web application that uses a straight database connection without the assistance of DAO layer such as Hibernate or Spring DAO, a connection to the database is made using something like the following:

```
public class  MyServiceBean {

    public void startProcess() {

        DataSource ds = (DataSource) new InitialContext().lookup("jdbc/ds1");
        con = ds.getConnection();

        //do something with connection
    }
}
```

The `DataSource` is simply an interface which the underlying application container implements in order to allow applications to get connections. In this way, a web application server can control the way connections are handed out as well as keep connections in a pool to be reused. The information about the data source, including the driver, host name, user name, password and database name are all setup on the web application server. The way they're configured varies depending on the server (most have a web administration console or XML configuration files), but all application servers provide an initial context containing references to these data sources for all running web applications.

If we simply tired to test this bean using a `main` function in its own stand-alone application like so:

```
public static void main(String[] args) {
    MyServiceBean b = new MyServiceBean();
    b.startProcess();
}
```

We would get the following exception:

```
javax.naming.NoInitialContextException: Need to specify class name in environment or system property, or as an applet parameter, or in an application resource file:  java.naming.factory.initia
```

```
        at javax.naming.spi.NamingManager.getInitialContext(Unknown Source)
        .....
```

In order to run code which looks up a `DataSource` this way in a stand-alone application, we must create our own implementation of a `DataSource` object as well as an `InitialContextFactory`, plus several intermediary objects, and then tell our currently running JRE to use our context object for all subsequent calls to `new InitialContext()`. Because we're running locally, we don't need to implement everything out of all these interfaces, just the bare minimum in order to provide `DataSource` and `Connection` objects.

First, we'll look at a very quick and dirty solution. In this solution, we declare new classes directly within the main function using the `final` keyword and hard-coding the driver and connection strings. This is a quick drop to simply test a single bean.

```
public static void main(String[] args) throws SQLException, ClassNotFoundException, NamingException
{
    final class LocalDataSource implements DataSource , Serializable {

            private String connectionString;
            private String username;
            private String password;

            LocalDataSource(String connectionString, String username, String password) {
                this.connectionString = connectionString;
                this.username = username;
                this.password = password;
            }

            public Connection getConnection() throws SQLException
            {
                return DriverManager.getConnection(connectionString, username, password);
            }

            public Connection getConnection(String arg0, String arg1)
                    throws SQLException
            {
                return getConnection();
            }

            public PrintWriter getLogWriter() throws SQLException
            {
                return null;
            }

            public int getLoginTimeout() throws SQLException
            {
                return 0;
            }

            public void setLogWriter(PrintWriter out) throws SQLException {}

            public void setLoginTimeout(int seconds) throws SQLException {}

    }

    final class DatabaseContext extends InitialContext {

        DatabaseContext() throws NamingException {}

        @Override
        public Object lookup(String name) throws NamingException
        {
            try {
                //our connection strings
                Class.forName("com.mysql.jdbc.Driver");
                DataSource ds1 = new LocalDataSource("jdbc:mysql://dbserver1/dboneA", "username", "xxxpass");
                DataSource ds2 = new LocalDataSource("jdbc:mysql://dbserver1/dboneB", "username", "xxxpass");

                Properties prop = new Properties();
                prop.put("jdbc/ds1", ds1);
                prop.put("jdbc/ds2", ds2);

                Object value = prop.get(name);
                return (value != null) ? value : super.lookup(name);
            }
             catch(Exception e) {
                 System.err.println("Lookup Problem " + e.getMessage());
                 e.printStackTrace();
             }
             return null;
        }

    }

    final class DatabaseContextFactory implements  InitialContextFactory, InitialContextFactoryBuilder {

        public Context getInitialContext(Hashtable<?, ?> environment)
                throws NamingException
        {
            return new DatabaseContext();
        }

        public InitialContextFactory createInitialContextFactory(
                Hashtable<?, ?> environment) throws NamingException
        {
            return new DatabaseContextFactory();
        }

    }

    NamingManager.setInitialContextFactoryBuilder(new DatabaseContextFactory());

    MyServiceBean b = new MyServiceBean();
    b.startProcess();
}
```

In this example we'll start from the bottom. Before we run our bean we see a call to `NamingManager.setInitialContextFactoryBuilder()`. This function sets the factory that will be called in our environment by all subsequent calls to `new InitialContext()` throughout our application. Underneath the hood of a web application server, this is one of the many things that happens well before a web application is loaded.

The `DatabaseContextFactory` is a class we create that not only serves as a `ContextFacotry` but also a `ContextFactoryBuilder` through appropriate interfaces. Thanks to interfaces, we can simplify these two functionalities into a single class. It's sole purpose is to return a `DatabaseContext`.

The `DatabaseContext` extends a regular `InitialContext` and we simply override the one function typically used by web server code, the `lookup()` function. It is here we can inject our own `LocalDataSource` objects, which return our `Connection` objects. On an actual web application server, the `DataSource` object would typically have some type of pooling mechanism that could return existing connections into a queue once the web application calls the `close()` function on them.

Although is is a decent quick solution, it's really unclean and isn't reusable without copying and pasting. It also ignores any environment parameters passed into the `InitialContext` and discards them. For a more permanent solution, each of the classes should be separated out and compatibility should be maintained with the environment properties passed in to our custom `Context` object.

For our clean and reusable solution, we're only going to have one class with public visibility. It's our factory class. All other classes will have default/package visibility because they shouldn't be instantiated independently outside of our factory. Let's start with the factory:

```
package org.penguindreams.db.local;
```

```
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.naming.spi.InitialContextFactory;
import javax.naming.spi.InitialContextFactoryBuilder;

public class LocalContextFactory {
        /**
         * do not instantiate this class directly. Use the factory method.
         */
        private LocalContextFactory() {}

        public static LocalContext createLocalContext(String databaseDriver) throws SimpleException {

                try {
                        LocalContext ctx = new LocalContext();
                        Class.forName(databaseDriver);
                        NamingManager.setInitialContextFactoryBuilder(ctx);
                        return ctx;
                }
                catch(Exception e) {
                        throw new SimpleException("Error Initializing Context: " + e.getMessage(),e);
                }
        }
}
```

In the above code, we're creating a new `LocalContext`. We're also initializing our JDBC driver. As with the previous example, the `NamingManager.setInitialContextFactoryBuilder` function ensures the new context we're creating will be given to all subsequent calls made to `new InitialContext()`. Also, as with the previous example, the `LocalContext` takes both the role of an `InitialContextFactory` and an `InitialContextFactoryBuilder` through the use of interfaces.

```
package org.penguindreams.db.local;

import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.naming.spi.InitialContextFactory;
import javax.naming.spi.InitialContextFactoryBuilder;

class LocalContext extends InitialContext implements InitialContextFactoryBuilder, InitialContextFactory {

        Map<Object,Object> dataSources;

        LocalContext() throws NamingException {
                super();
                dataSources = new HashMap<Object,Object>();
        }

        public void addDataSource(String name, String connectionString, String username, String password) {
                this.
                dataSources.put(name, new LocalDataSource(connectionString,username,password));
        }

        public InitialContextFactory createInitialContextFactory(
                        Hashtable<?, ?> hsh) throws NamingException {
                dataSources.putAll(hsh);
                return this;
        }

        public Context getInitialContext(Hashtable<?, ?> arg0)
                        throws NamingException {
                return this;
        }

        @Override
        public Object lookup(String name) throws NamingException {
                Object ret = dataSources.get(name);
                return (ret != null) ? ret : super.lookup(name);
        }
}
```

In the above example, we also see that we've allowed for some default behavior. For instance, properties that are given to initialize our `LocalContext` are stored in our local `HashMap`. If a lookup fails, we call the parent's lookup method as well. The important function we add above is the `addDataSource()` method which creates new `LocalDataSource` to be looked up by the passed in `name` argument. Finally we have the `LocalDataSource` itself.

```
package org.penguindreams.db.local;

import java.io.PrintWriter;
import java.io.Serializable;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import javax.sql.DataSource;

class LocalDataSource implements DataSource , Serializable {

        private String connectionString;
    private String username;
    private String password;

    LocalDataSource(String connectionString, String username, String password) {
        this.connectionString = connectionString;
        this.username = username;
        this.password = password;
    }

    public Connection getConnection() throws SQLException
    {
        return DriverManager.getConnection(connectionString, username, password);
    }

        public Connection getConnection(String username, String password)
                        throws SQLException {return null;}
        public PrintWriter getLogWriter() throws SQLException {return null;}
        public int getLoginTimeout() throws SQLException {return 0;}
        public void setLogWriter(PrintWriter out) throws SQLException { }
        public void setLoginTimeout(int seconds) throws SQLException {}
}
```

As you can see, many of the functions for the `DataSource` have been left unimplemented. We've only implemented enough functionality to get lookups working and to create simple, non-pooled connections for the end client. So our main function should now look something like the following:

```
public static void main(String[] args) {

    LocalContext ctx = LocalContextFactory.createLocalContext("com.mysql.jdbc.Driver");
    ctx.addDataSource("jdbc/js1","jdbc:mysql://dbserver1/dboneA", "username", "xxxpass");
    ctx.addDataSource("jdbc/js2","jdbc:mysql://dbserver1/dboneB", "username", "xxxpass");

    MyServiceBean b = new MyServiceBean();
```

```
    b.startProcess();
}
```

The call to `LocalContextFactory.createLocalContext()` initializes our environment with the `LocalContext` as the `InitialContext`. Then we can add the `DataSource` objects we need later using the `addDataSource()` method. All of our data objects are kept within their own packages and have limited visibility to ensure they can only be instantiated and fully initialized using the factory method.

There are some limitations to this *clean* implementation. For one, you can only use one type of database depending on what driver you specify with the `LocalContextFactory`. Also, not all of the `DataSource` methods are fully implemented, so you may run into problems in environments that depend on other functions and more complex implementations.

Still, the above code will work in a testing environment and, in a pinch, you can use the final classes used at the beginning of this tutorial directly in a `main` function for some quick and dirty testing. If you need a more complete implementation, the clean version of the code is an excellent starting point to begin building a full local implementation of your own custom `DataSource` objects.

Tags: data sources, EJB, java, JDBC

This entry was posted on Tuesday, May 11th, 2010 at 9:03 am and is filed under Blog. You can follow any responses to this entry through the RSS 2.0 feed. You can skip to the end and leave a response. Pinging is currently not allowed.

13 Responses to “Running Beans Locally that use Application Server Data Sources”

- *Gregory Smith* Says:
  February 23rd, 2012 at 5:21 pm

  Dude, you should be paid real money for this tip. I have needed such a solution for years and you just helped me create a test harness for my Spring Batch database apps.

  Many thanks,

  Greg

- *Deepak* Says:
  July 6th, 2012 at 4:42 am

  Dear Sir,

  Thanks a lot for this article. I had gone through tons of stuff regarding “JNDI in J2SE“(including oracle tutorials). Everyone talking in riddles. Finally i found your post and its working. Hats off to you for such a nice and concise article on a very complex topic.

- *Anand* Says:
  August 14th, 2012 at 6:57 am

  You are a eyeopener for me, thanks for the articlet, simply suberb

- *doobie* Says:
  November 5th, 2012 at 1:08 pm

  Thanks a lot, that’s what I was looking for ! Good Job

- *Mohammady* Says:
  November 29th, 2012 at 10:59 am

  Dude Thanks a ton, I was thinking on the same lines as I needed to locally test spring/hibernate/jbpm app outside of WLS.

  You saved my day!!

- *VIjay* Says:
  December 2nd, 2012 at 7:44 am

  This is what i’m looking for, thanks dude.

- *Jaehak Lee* Says:
  February 22nd, 2013 at 10:15 pm

  Thank you very much.!!
  This is What I am looking for.
  It’s very helpfull for me.

- *m srinivas charan* Says:
  March 8th, 2013 at 4:06 am

  thank u very very much…..great

- *Yug Suo* Says:
  April 27th, 2013 at 3:56 am

  Can we configure a global jndi database to solve this problem ?

- *Firionel* Says:
  September 6th, 2013 at 9:28 am

  Brilliant – cheers.

  This just totally made my day.

- *Sudhakar* Says:
  September 24th, 2013 at 11:55 am

  Thank you for the great article. I have a question that I hope you can help me understanding it.

  If I create a Datasource on a server that access the database, is there any way I can access that DataSource instead of creating my own by passing connectionString, driver name, userid, password? By doing this, I think we are not really using the Datasource that was created on the server, instead creating our own. If the purpose of this article is to explain how to create a DataSource, can I ask you to help me how can I use the DataSource that was created on the server?

  I am using RAD7.5.4 with WAS6.1 test server.

  Thank you!
  Sudhakar

- *Sameer Siddiqui* Says:
  September 27th, 2013 at 10:59 pm

  Many Thanks for the wonderful insight and solution on using JNDI / J2SE this helped me test my Model without any containers. Brilliant

- *Aspasia* Says:

November 28th, 2013 at 11:15 am

You are a star! Was looking for it for so long!

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Type the text

Privacy & Terms

Submit Comment