Get Involved

java-net Project
Request a Project
Project Help Wanted Ads
Publicize your Project
Submit Content

Get Informed

About java.net
Articles
Weblogs
News
Events
Also in Java Today
java.net Online Books
java.net Archives

Get Connected

java.net Forums
Wiki and Javapedia
People, Partners, and Jobs
Java User Groups
RSS Feeds

Search

Web and Projects: »
Online Books: »
Advanced Search

Project Wonderland v0.5: Synchronizing state across clients (Part 4)

by Jordan Slott (jslott@dev.java.net)

Purpose

In this tutorial, you will complete the simple new cell you have created in Part 1, Part 2, and Part 3 of this tutorial series by enabling it to synchronize its state among many Wonderland clients. You will modify your shape cell to communicate when the user clicks on the shape to change its form (cube vs. sphere) back to the server-side object. The server-side object updates its state in a thread-safe manner, and communicates the change back to all of the clients.

This tutorial is designed for Project Wonderland User Preview 2.

You can find the entire source code for this module, including code for future tutorials in the "unstable" section of the Project Wonderland modules workspace. For instructions on downloading this workspace, see Download, Build and Deploy Project Wonderland v0.5 Modules.

Expected Duration: 45 minutes

Prerequisites

Before completing this tutorial, you should have already successfully completed Part 1, Part 2, and Part 3 of this tutorial series. You will be extending the functionality you implemented there.

You should also be familiar with programming Project Darkstar: it is the middleware technology upon which Project Wonderland is built that handles client-server communication and makes it easy to write server-side objects to manage their state in a thread-safe manner. Visit the Project Darkstar website--it is best to download the distribution and read the server-side tutorial document.

High-level Design

From a high-level, here is what you will implement in this tutorial to enable your cell type for a multi-user environment:

1. Communicate the new shape type from the client-side ShapeCell class to the server-side ShapeCellMO class when the user clicks on the shape
2. Update the shape type state stored by the ShapeCellMO class
3. Communicate the new shape type to all other clients

Creating a new message: ShapeCellChangeMessage

Before the client can communicate any changes in the type of shape when the user clicks on the shape, you need to create a new class to communicate this message.

In the **org.jdesktop.wonderland.shape.common** package, create a new class named ShapeCellChangeMessage.java. Add the following import statements near the top of the file:

```
import org.jdesktop.wonderland.common.cell.CellID;
import org.jdesktop.wonderland.common.cell.messages.CellMessage;
```

Next, have your class extend the CellMessage class as follows:

```
public class ShapeCellChangeMessage extends CellMessage {
```

The CellMessage class is the base class for all messages passed between the client and server cell classes in Project Wonderland. There are no methods in CellMessage that you must override, however.

This message communicates the new shape of the cell to be displayed on all clients, so add a field to your class to store this new type (as a string), along with setter and getter methods:

```
    private String shapeType = null;

    public String getShapeType() {
        return this.shapeType;
    }

    public void setShapeType(String shapeType) {
        this.shapeType = shapeType;
    }
```

Next, implement the following constructor. As its first argument, it takes a CellID object: this is necessary so that when the message is received by the Wonderland server, it knows to which cell object it should dispatch the messsage. Each cell's ID is unique and is

assigned automatically when it is created by the Wonderland system.

```
    public ShapeCellChangeMessage(CellID cellID, String shapeType) {
        super(cellID);
        this.shapeType = shapeType;
    }
```

Communicating the Shape Change to the Server

Next, you will modify your ShapeCell class to send a message to the server when the user clicks on the shape to change its shape type (cube/sphere). We send this message over a special channel automatically created for you by the system that is a link between your client-side Cell class (ShapeCell) and server-side class (ShapeCellMO).

Locate the **commitEvent()** method in your MouseButtonListener inner class in ShapeCell.java. After the end of this method, insert the following two lines to create a new ShapeCellChangeMessage object with the new shape type and send the message to the server:

```
            ShapeCellChangeMessage msg = new ShapeCellChangeMessage(getCellID(), shapeTy
            sendCellMessage(msg);
```

The **Cell.sendCellMessage()** method is a convenience method that sends a message (of subclass CellMessage) on the cell's channel. This has the effect of immediately updating the shape type visually on the client, so that the user gets immediate feedback and then sends a message to the server with the new shape. The **getCellID()** method is defined by the Cell superclass and simply returns the unique ID of the cell that is automatically assigned to the cell when it is created.

You will also need to add the following import statements to near the top of your ShapeCell.java file:

```
import org.jdesktop.wonderland.modules.shape.common.ShapeCellChangeMessage;
```

Synchronizing the State of the Cell on the Server

Your server-side cell class, ShapeCellMO, now needs to do the following:

1. Register to receive the ShapeCellChange message
2. Receive the message from the client
3. Update its state based upon the new shape type sent from the client
4. Inform all other clients of the new shape type

Fortunately, because Wonderland is built on top of Project Darkstar, synchronizing the state among many different concurrent clients is made easy. You simply need to register a listener to receive the event from the client and update the member variable in your ShapeCellMO class. First, you will define the event listener, ShapeCellMessageReceiver.

The ShapeCellMessageReceiver class should be a static inner class of ShapeCellMO, defined (in part) as follows:

```
    private static class ShapeCellMessageReceiver extends AbstractComponentMessageReceiv
        public ShapeCellMessageReceiver(ShapeCellMO cellMO) {
            super(cellMO);
        }
        public void messageReceived(WonderlandClientSender sender, WonderlandClientID cl
        }
    }
```

A few important words are in order: while most of the Wonderland APIs do not expose the underlying Darkstar middleware layer, the Darkstar transaction architecture does have an impact on how you define classes to receive messages on communication channels. Specifically, you are **not** permitted to use *non-static inner classes or anonymous classes* as message receivers for the channel. Here you use a private static class instead. You extend the AbstractComponentMessageReceiver class, which helps manage the cell and the channel component associated with this message receiver.

Whenever a ShapeCellChangeMessage is received, it calls the **messageReceived()** method in this class. The **messageReceived()** method right now is empty -- you will fill in its details shortly.

Now's probably a good time to add some imports to the top of your ShapeCellMO class. Of course, it is always a good idea to have your IDE "fix" your import statements for you.

```
import org.jdesktop.wonderland.server.cell.AbstractComponentMessageReceiver;
import org.jdesktop.wonderland.server.comms.WonderlandClientSender;
import org.jdesktop.wonderland.common.cell.messages.CellMessage;
import org.jdesktop.wonderland.modules.shape.common.ShapeCellChangeMessage;
import org.jdesktop.wonderland.server.cell.ChannelComponentMO;
```

Next, you'll implement the details of the **messageReceived()** method. It will accomplish two things: update the state of the **shapeType** member variable to reflect the new shape and communicate the new shape type to all of the clients. Since Project Wonderland is a multi-user environment, there may be several different users who click on their shape to update the shape type at the same time. In a typical multi-user environment, you must make sure updates to the state type is synchronized with all other possible updates. This is often a very tricky task!

Fortunately, the Project Darkstar infrastructure makes this really simple. All updates to the state of server-side objects happen

within the context of a *transaction* -- either the state update happens at once without conflicting with other requests to update the state of the object, or it does not happen at all. The Project Darkstar infrastructure manages the resource contention for you: it notes what objects you update and when the transaction completes and commits its changes, it does so in an atomic fashion. It knows ShapeCellMO is such an object to manage in a transaction because (in Part 1) you had it implement the Darkstar ManagedObject interface by extending the CellMO class.

When a message is delivered to your server-side cell class, it automatically happens within the context of a transaction: all you need to do is update the state you wish and when the method completes, Project Darkstar will commit any changes you made atomically and in a multi-user safe manner.

So, in your **messageReceived()** method, the first line will be:

```
            ShapeCellMO cellMO = (ShapeCellMO)getCell();
```

This fetches the cell class associated with the message receiver. You will need this below. Next, cast the message received to type ShapeCellChangeMessage and update the shape type in the ShapeCellMO object:

```
            ShapeCellChangeMessage sccm = (ShapeCellChangeMessage)message;
            cellMO.shapeType = sccm.getShapeType();
```

These two lines are all that's required to insure that the state of this cell that is shared among many users is updated in a safe manner. Note that you set the **shapeType** member variable on the cell object we obtained from the AbstractComponentMessageReceiver. Since this method is an inner class of ShapeCellMO, you can set the variable on the **cellMO** object directly, rather than needing a *public* method on ShapeCellMO to set the shape type.

You next need to register this message receiver on the communications channel of the Cell. To do this, you will override the **setLive()** method defined by CellMO, as follows:

```
    @Override
    protected void setLive(boolean live) {
        super.setLive(live);

        ChannelComponentMO channel = getComponent(ChannelComponentMO.class);
        if (live == true) {
            channel.addMessageReceiver(ShapeCellChangeMessage.class,
                (ChannelComponentMO.ComponentMessageReceiver)new ShapeCellMessageReceive
        }
        else {
            channel.removeMessageReceiver(ShapeCellChangeMessage.class);
        }
    }
```

If the cell is being made 'live' then we register the message receiver to receive messages of type ShapeCellChangeMessage and remove the receiver if the cell is being made 'unlive'. Note that the system automatically creates a Cell channel for you; it is guaranteed to exist when your **setLive()** method is invoked.

Finally, send a message to all of the clients connected of the new state using the channel component. Note that you'll send a message back to the same client which sent the message in the first place. You'll want that client to disregard that message. In order to do this, you'll pass the unique ID of the client back in the message. The server must generates this unique ID so that clients cannot "spoof" other clients in this regard. Add the following line to your **messageReceived()** method:

```
            cellMO.sendCellMessage(clientID, message);
```

The **CellMO.sendCellMessage()** is a convenience method that sends a message on the communication channel automatically created for you for the Cell.

Improper use of non-static or anonymous inner classes

As mentioned before, non-static inner classes may not be used for message receivers. To be concrete, in Wonderland the following design pattern is **incorrect** and should never be used for message receivers:

```
    /* Non-static inner class not permitted! */
    class ShapeCellMessageReceiver extends AbstractComponentMessageReceiver {
        public ShapeCellMessageReceiver(ShapeCellMO cellMO) {
            super(cellMO);
        }
        public void messageReceived(WonderlandClientSender sender, WonderlandClientID cl
            ShapeCellChangeMessage sccm = (ShapeCellChangeMessage)message;
            shapeType = sccm.getShapeType();   /// NOT CORRECT!
        }
    }
```

The following use of an anonymous inner class is also **not** correct:

```
      /* Anonymous class not permitted! */
      channel.addMessageReceiver(ShapeCellChangeMessage.class,
```

```
                new AbstractChannelMessageReceiver() {
                    ....
                }
    );
```

Listening for ShapeCellMessage on the Client

There is one final step that completes the loop: your ShapeCell client-side class must also listen for messages sent to it that inform it of a new shape type. Here, your ShapeCell class must implement an inner class to receive messages much like you did in ShapeCellMO. First, add the following import statements to the top of your ShapeCell.java file:

```
import org.jdesktop.wonderland.client.cell.ChannelComponent;
import org.jdesktop.wonderland.client.cell.ChannelComponent.ComponentMessageReceiver;
import org.jdesktop.wonderland.common.cell.messages.CellMessage;
```

Next, your ShapeCell class must register to listen for messages of type ShapeCellChangeMessage.class. You can only add a listener on the cell channel after it has been created for you by the system: the cell channel is guaranteed to exist when the system calls your **setStatus()** method (much like you did for registering a listener for mouse click events in **setStatus()**).

In the if-clause for the RENDERING state, add the following

```
        ChannelComponent channel = getComponent(ChannelComponent.class);
        channel.addMessageReceiver(ShapeCellChangeMessage.class, new ShapeCellMessageRece
```

This adds a listener for messages over the cell communication channel of type ShapeCellChangeMessage.class. You can remove this listener when the status of the Cell goes back to DISK (when it is no longer visible in the world). In the if-clause for the DISK state, add the following:

```
        channel.removeMessageReceiver(ShapeCellChangeMessage.class);
```

Implement the following inner class, ShapeCellMessageReceiver, much like you did on the server. Since the client is not part of the server-side Darkstar transaction mechanism, you are free to use any kind of inner class you like on the client, including non-static and anonymous inner classes:

```
    class ShapeCellMessageReceiver implements ComponentMessageReceiver {
        public void messageReceived(CellMessage message) {
            ShapeCellChangeMessage sccm = (ShapeCellChangeMessage)message;
            if (!sccm.getSenderID().equals(getCellCache().getSession().getID())) {
                shapeType = sccm.getShapeType();
                renderer.updateShape();
            }
        }
    }
```

The **messageReceived()** method simply casts the message to a ShapeCellChangeMessage message, fetches the new shape type, sets the **shapeType** member variable to the new type, and asks the renderer to redraw the shape. The **if** statement in the **messageReceived()** method checks the ID stored in the message (that you set in the server-side class) and compares it to the unique ID of its client, which is obtained via **getCellCache().getSession().getID()**.

Running Your New Shape Cell

You must now recompile your module and re-deploy it into your instance of Wonderland. Please follow the instructions in the previous tutorials on how to do this.

- Wonderland Web-Based Administration Guide
- Project Wonderland v0.5: Working with Modules
- Project Wonderland v0.5: Developing a New Cell (Part 1)

You'll also want someone else on another machine to run the same client. (Note that you will both have to use the same client code -- you can launch clients via Java Webstart for this purpose).

Whenever you click on the box or sphere, it should change to the other shape type. The other user should see the shape change type too!

Summary

In this four part tutorial series, you learned how to extend Wonderland by creating a new kind of cell. You set up the project infrastructure, drew a basic object on the screen, and then used the client side asset manager to load a texture for the shape. Finally, you learned how to make your new cell synchronize its state among many Wonderland clients.

indicates your agreement to be bound by these Terms of Participation.

Revision r1 - 2009-12-25 - 03:27:22 - Main.jslott
Parents: WebHome > ProjectWonderland