Qooxdoo Server-Objects

The server-objects contrib. integrates Java POJO objects on the server with the Qooxdoo client application in a natural and seamless manner. Objects on the server support properties and events, and can fine tune how data is transferred and cached with annotations. Objects can be passed by value between the client and server which allow (for example) server objects to be bound to a Qooxdoo form without requiring any boiler plate code for communication.

The objects passed to the client are created as actual Qooxdoo objects, in that a class and interfaces are defined with calls to qx.Class.define() etc so that the object model on the client exactly matches the interfaces defined on the server and all the normal OO rules apply.

The methods in the generated classes trigger synchronous calls to the server object, and any return value is seamlessly translated back into a Qooxdoo JS object; the Qooxdoo JS objects can be passed back to the server as parameters to a method call, where they are translated back into the original Java objects before reaching your code.

Annotations are used to define properties on the object server and declare how they are translated to the client (EG on demand vs pre-fetched, or wrapped arrays on the client); changes to property values are queued and copied to the other side when a synchronous event occurs.

The only requirement on the server side is that the objects implement a marker interface– but this is not necessary for method return types or parameters. If a server method returns a value which does not support the marker interface it will be serialised to JSON and sent anyway; similarly, an arbitrary JSON object can be sent to the server as a parameter providing that the receiving parameter is suitable (E.G. a Collection or a Map).

Quick Start Guide

This example uses the “explorer” demo Qooxdoo app and the com.zenesis.qx.remote.explorer package.

Defining a Java POJO

There is only one basic requirement when defining a POJO on the server – the class or interface must implement com.zenesis.qx.remote.Proxied, after which your methods can be called (synchronously) from the client. A more useful object will add properties to the class by using the Properties and Property annotations; for example:

```
@Properties({
    @Property(value="name",event="changeName")
})
public class ServerFile implements Proxied {
    private String name;
    public String getName(){
          return name;
    }
```

```
    public boolean rename(String name) {
          ...
    }
}
```

The Property annotation tells the QSO library about the property “name”, that it fires a “changeName” event whenever it is modified, and when QSO copies a ServerFile object from the server to the client, the property value must be delivered at the same time and then cached on the server. If the server changes the value of the property, it will fire a “changeName” event and QSO will relay the new value to the client.

QSO knows that the “name” property is read only because there is only a getXxxx method, but you can have read-write properties too:

```
@Properties({
    @Property(value="name",event="changeName"),
    @Property("readOnly")
})
public class ServerFile implements Proxied {
    private boolean readOnly;
    public boolean isReadOnly() { return readOnly; }
    public void setReadOnly(boolean readOnly) {
          this.readOnly = readOnly; }
```

In this example, the readOnly property will be defined on the client and the server and QSO will make sure that changes on the client will be copied to the server and vice-versa.

Properties On Demand

Some properties you only want to get on demand, but once they have been fetched the value should be cached on the client. For example:

```
@Properties({
    @Property(value="name",event="changeName"),
    @Property("readOnly"),
    @Property(value="children", onDemand=true, event="changeChildren")
})
public class ServerFile implements Proxied {
    private ServerFile[] children;
    public ServerFile[] getChildren() {
          if (children != null || !file.isDirectory())
                return children;

          File[] files = file.listFiles();
          [ ... snip ... ]
          return children;
    }
```

Note the “onDemand=true” that’s been added to the @Property definition; the getChildren() method will only be called once by the Qooxdoo client and the result is then cached on the client for future use.

Connecting the Servlet to the Client

Accessing objects on the server is just like any other method call (static methods are not supported) and to call a method you need an object. In QSO, the first object is called a “bootstrap” and is

created on demand by QSO; the client application uses the bootstrap object to get everything else. Bootstrap objects are nothing special – they must support the Proxied interface and have a no-args constructor.

Your Java Servlet needs to connect requests coming from the QSO library on the client to the QSO library on the server. First, choose a URL that is exclusively for QSO to use for your application (e.g. "/explorerServlet/ ajax") and then add the following code to your doPost method:

```
ProxyManager.handleRequest(request, response, FileExplorer.class,
"fileExplorer");
```

The first two parameters are the request and response objects passed to doPost() by the servlet container; the third parameter is the bootstrap class for your application and the fourth is a unique name for the application (it's used as a key in the HttpSession so make sure it's unique!)

In your client app, pass the URL to ProxyManager and then you can call getBootstrapObject() to get an instance of your bootstrap object:

```
var manager =
  new com.zenesis.qx.remote.ProxyManager("/explorerServlet/ajax");
var boot = manager.getBootstrapObject();
```

In the "explorer" sample client app, the source code is copied from the Qooxdoo example for virtual trees with a minor addition to get the tree nodes on demand:

```
tree.addListener("treeOpenWhileEmpty", function(evt) {
    var node = evt.getData();
    var children = node.serverFile.getChildren();
    for (var i = 0; i < children.length; i++) {
          var file = children[i];
          var nodeId;
          if (file.getFolder())
                nodeId = dataModel.addBranch(node.nodeId,
                      file.getName(), null);
          else
                nodeId = dataModel.addLeaf(node.nodeId,
                      file.getName(), null);
          tree.nodeGet(nodeId).serverFile = file;
    };
}, this);

// Create the root node
var rootNode = tree.nodeGet(dataModel.addBranch(null, "Desktop",
    false));
rootNode.serverFile = boot.getRoot();
```

The client code above is pure Qooxdoo – there is no boiler plate code for talking to the server. The "treeOpenWhileEmpty" event creates new nodes in the tree according to the children returned by ServerFile.getChildren(), and the root node is created by calling the bootstrap object's getRoot().

Working Examples

QSO includes two sample applications – "demoapp" which aims to test every aspect of QSO and report "All tests passed!" at the end, and "explorer" which is a simple file explorer that can navigate a tree of files, loaded on demand and bound to a Qooxdoo form.

Further Reading

Arrays

QSO recognised two types of array – native arrays (i.e. Object[]) and wrapped arrays (i.e. ArrayList on the server and qx.data.Array on the client) – and will transparently switch between either if required.

By default, QSO will create arrays on the client in the same form as the server – i.e. if the server has a native array so will the client and if the server has ArrayList, then the client will have qx.data.Array. You can change this behaviour with the "array" attribute of Property:

```
@Property(value="myNativeArray", array=Remote.Array.WRAP),
@Property(value="myArrayList", array=Remote.Array.NATIVE)
...
String[] myNativeArray;
ArrayList<String> myArrayList;
```

In the above example, myNativeArray will be created on the client by wrapping it with qx.data.Array, whereas the myArrayList will be implemented as a native Javascript array.

Events and Detecting Changes on the Server

QSO keeps properties values synchronised between client and server in as efficient a manner as possible but this means being able to add event listeners to detect changes. Qooxdoo does this already on the client and so we need an equivalent for Java on the server.

The ProxyManager class provides static methods to simplify notifying property changes and firing events, which will be relayed to the client.

Here's an excerpt from ServerFile.rename() which changes the "name" property:

```
public boolean rename(String name) {
    if (name.equals(this.name))
          return true;

    // Validation code removed....

    // Change the property
    String oldName = name;
    this.name = name;
    ProxyManager.propertyChanged(this, "name", oldName, name);
    return true;
}
```

ProxyManager also supports fireEvent() and fireDataEvent()

Creating Server Objects From The Client

When the server app passes a Proxied object to the client, it sends a definition of the class to the client and qx.Class.define() is used to define a parallel class on the client. Because these are normal

Qooxdoo classes your client application can create an instance of that class as and use it as any other object.

When you pass one of these client objects to the server (e.g. the client passes the object as a parameter to a method) the QSO library will create a corresponding object on the server and set its properties to match the client.

Thereafter, the object is kept synchronised on both sides, just like any other server object.

Interfaces and Class Heirarchy

From the examples above you'll see that when you send a POJO which has no super class or interfaces (except Proxied), then QSO will look to that class to get the list of properties, events and methods to create on the client.

QSO also supports arbitrarily complex class hierarchies with interfaces and/or superclasses. The only requirements are that each class or interface must implement Proxied, and there must be no conflicting property or method names (a conflict is where two or more properties/methods have the same name but a different definition, e.g. two properties with the same name but different type). Each interface and class will be replicated on the client so that when objects are copied from one side to the other all the normal inheritance properties apply.

By default, if a class implements an interface that extends Proxied, then only properties/methods from the interface will be copied; you can change this by adding the AlwaysProxy annotation.

Annotations Reference Guide

AlwaysPresent

This annotation tells QSO that the class or interface already exists on the client – perhaps because it has been hand coded so it can support a more sophisticated range of methods and properties that are only applicable to the client. When QSO needs to create an instance of the class on the client it will be using your hand written class rather than one copied from the server.

AlwaysProxy

By default methods are proxied or not depending on context, e.g. if a class implements an interface that extends Proxied the classes method is not copied. The AlwaysProxy annotation can be added to a class or individual methods to override the default.

DoNotProxy

This will prevent a method from being proxied

Event

Declares that a class can fire an event with a given name. The attributes are:

value (the default) – the name of the event that can be fired.

sync – specifies whether the event should be relayed immediately (i.e. synchronously) or queued until the next trip to the server is required; the default is to queue the event.

data – the type of the data in the event, if fireDataEvent is used; the default is Object.class.

Events

Used only to collect a list of Event annotations

ExplicitProxyOnly

This is applied to a class and is the opposite to AlwaysProxy and will change the default to not override when otherwise QSO would have copied the method(s), i.e. it requires the method to be marked with AlwaysProxy

Function

This is applied to a method to customise how the method is proxied. If the method returns an array, the array property says whether to wrap the array at the client with qx.data.Array or to have a native array. The "arrayType" property gives the component type of the array but is only necessary if the method returns a collection (if the method returns a native array then the component type is used).

Properties

Used only to collect a list of Property annotations.

Property

Used to declare a property that can be copied between client and server and to customise how that it done.

value (the default) – specifies the name of the property; reflection is used to find either a public field or getXxxx/setXxxx/isXxxx methods to read and write the property value.

array – if the property is an array, this specifies whether to wrap the array on the client with qx.data.Array or whether to provide it as a native array; by default it follows the server definition.

arrayType – if the property is a Collection the arrayType value must be used to set the component type; this is only used when deserialising on the server.

sync – specifies whether changes to the property should be relayed immediately (i.e. synchronously) or queued until the next trip to the server is required; the default is to queue the property change.

onDemand – specifies whether the property is delivered with the object or whether to wait and only get the value when the client app first requires it and then cached on the client; the default is false.

readOnly – specifies whether the property is read only, even if there is a setXxxx() method; the default is to auto-detect.

event – the name of the event that will be fired when the property changes; the default is that no event is fired.

nullable – the value used for the “nullable” attribute when defining the Qooxdoo property, i.e. whether the property allows null values; the default is true.

How it works

By default, QSO only proxies classes and interfaces which directly extend the marker interface Proxied; when you try and serialise a Proxied object to JSON, the serialiser writes a unique ID and a class name to the output instead of a normal JSON serialisation.

The first time an object of a given class is sent to the server, the serialiser will also send a definition of the class, including its methods and the Proxied-derived interfaces and superclasses. The class definition is also sent if a method returns objects of that class or accepts them as parameters.

The serialiser uses a class called ProxySessionTracker to identify which classes have previously been defined in this session, and outputs the definition if necessary.

Even though the object model is kept identical between the client and the server, the use case will often vary and one way to do this is to implement the client class by hand; implementing remote method calls is a single line of Javascript. You then declare server classes as already available on the client by specifying the @AlwaysPresent annotation on the server class and the library will take care of the rest.

Is it really ANY kind of object?

Yes – in theory. For security concerns it is usually unwise to expose native server objects, at least not fully (you might want to show a list of files but you probably wouldn’t want to expose the whole of java.io.File ☺).

QSO automatically exposes classes and interfaces which extend the Proxied interface, and it only considers the methods defined as part of that interface/class as suitable for proxying to the client. You can override this behaviour if you want to by creating your own instance of ProxyType and registering it with ProxyTypeManager.

ProxySessionTracker

There is exactly one ProxySessionTracker for each client using a Qooxdoo Application and the ProxyManager.handleRequest static method takes care of this provided you give it a unique name for your application.

ProxyManager

ProxyManager is used to hook up the Jackson JSON processor to the ProxySessionTracker for the application; it must be able to identify the ProxySessionTracker instance for the application. The current ProxySessionTracker is stored in a ThreadLocal variable in ProxyManager and if you manager your own tracker you must surround calls to QSO with ProxyManager.selectTracker and ProxyManager.deselectTracker.

Dynamic vs Static Class Definitions

QSO uses reflection to get the class definition and assumes that very few (or none) classes already exist on the client and each time an object is serialised it checks whether the class definition has already been shipped and if necessary sends that along with the object.

What's good about this is that it gives a very fluid approach to developing an application because there is only one class definition to maintain, therefore it is always up to date and does not depend on an Ant task or similar (which is good if you use an IDE like Eclipse instead of Ant).

What's bad is that class definitions are being serialised in every session and then sent down the wire in an un-optimised form. It should be trivial to generate class definitions to disk and have them compiled by the generator for a production build, although there is currently no tool to do this.

Issues

Arrays

Arrays are wrapped on the client with qx.data.Array so that changes can be detected, but unfortunately the information in the event is not always enough to reconstruct the changes. The Java ArrayList does not provide event notifications at all, so changes on the server are not replicated anyway.

Roadmap

- Fix arrays for efficient updates between client and server (both directions)
- Server XML bindings
- Generate class definitions for compilation by the generator