Enabling R packages for web or grid services

Martin T. Morgan*, Nianhua Li, Seth Falcon,
Robert Gentleman,

30 November, 2006, 20 March, 2007

1 Preliminaries

1.1 Prerequisites

RWebServices and associated software must be installed; see the accompanying documentation "Installing and testing RWebServices and enabled packages".

You must have a valid R package, including NAMESPACE file. See the Writing R Extensions manual.

All complex objects to be translated to Java *must* be either primitive types (e.g., numeric, character) or S4 classes.

2 Creating Java templates

2.1 TypeInfo

Add type information to your functions.

1. Include TypeInfo as a 'Depends' line in the DESCRIPTION file.

2. Provide `typeInfo` for each method to be exposed. From the caDNAcopy package, an example is:

```
> typeInfo(caDNAcopy) <-
+   SimultaneousTypeSpecification(
+     TypedSignature(dnacopyAssays= "DNAcopyAssays",
+                    dnacopyParameter="DNAcopyParameter"),
+     returnType="DerivedDNAcopySegment")
```

   Provide this information within the package, in a '.R' file after the corresponding function (caDNAcopy) has been defined. See documentation and vignettes in the TypeInfo package for detail.

*Fred Hutchinson Cancer Research Center, 1100 Fairview Ave. N., PO Box 19024 Seattle, WA 98109

3. Install the package, e.g.,

   ```
   R CMD INSTALL --clean <pkg>
   ```

   where <pkg> is the name of your package. This can also be done from within R using `install.packages` or other means.

2.2 Unpack ant scripts

Unpack ant scripts with the R `unpackAntScript` command, or at the command line with

```
R -e "library(RWebServices); unpackAntScript('~/tmp/<pkg>')"
```

where `~/tmp/<pkg>` is the path to a temporary directory.

2.3 Create Java templates

There are several ways of proceeding. One way is to use `createMap` from within R. A second way is to change to the directroy where the ant scripts were unpacked, and evaluate

```
cd ~/tmp/<pkg>
ant -Dpkg=<pkg> map-package
```

(`~/tmp/<pkg>` is the directory where the ant scripts were unpacked). Both methods create a directory hierarchy `src/`, and usually `test/src`.

Sometimes additional Java templates maybe required for extra R data types. Suppose your function returns a *list* of *DerivedDNAcopySegment*. Your type information only shows `returnType="list"`, but you need the Java templates of *DerivedDNAcopySegment*. If you use `createMap` within R, use argument `extraClasses`. If you use the ant scripts, set the property `extra.classes` in `~/tmp/<pkg>/RWebServicesTuning.properties` to `DerivedDNAcopySegment`. You can also specify multiple R data types as extra classes in a comma delimited character string.

3 Writing and running tests

3.1 Writing test code – data

The files

```
test/src/org/bioconductor/rserviceJms/worker/RWorkerDataTest.java
test/src/org/bioconductor/rserviceJms/worker/R/*.R
test/src/org/bioconductor/rserviceJms/worker/Data/*.data
```

contain skeletons to help generate Java and R components for testing data transfer between R and Java. Templates are established for tests from Java to R for all function arguments, and from R to Java for all return values. If any extra classes are specified, their tests are established in both directions.

The Java code for testing uses the JUnit framework. A typical method starts with

```
 @Ignore("please initialize data")
 @Test
public void TestDNAcopyParameterToR() throws Exception {
    org.bioconductor.packages.caDNAcopy.DNAcopyParameter
        inputVal = null;
    inputVal = new ...
    String rScript =
        getClass().getResource("R/DNAcopyParameterData.R").getFile();
    String rVariable = "DNAcopyParameterData";
    assertTrue(myService.mockJava2R(inputVal, rScript, rVariable));
}
```

The first two lines are directives for JUnit. The test framework will arrange to pass `inputVal` to R, and use the value of the variable `rVariable` in `rScript` to assess whether the data transfer is successful. The developer needs to customize `inputVal` and the source file in the `test/src` hierarchy). Comment `@Ignore` to enable the test.

Serialized data instances can be added to the `Data` directory. Brave users can even render serialized Java data instances from R data instances. Save R objects into binary files, and put them in one directory, say `<data_dir>`, and then evaluate:

```
cd ~/tmp/<pkg>
ant create-data -Daction=load -Ddata.dir=<data_dir>
```

The ant task transfers those R objects into Java objects and saves them into binary files in the same directory. You can then use the serialized Java data in the test. This task requires the R to Java converts of the R objects. The R to Java converts are not created for function arguments. So PLEASE make sure your R objects are either a function return type or an extra class. An alternative task

```
ant create-data -Daction=data -Ddata.name=<dataset_name> \
        -Ddata.dir=<data_dir>
```

invokes R function `data` with argument `<dataset_name>`, and saves the serialized Java data in `<data_dir>`. The default `<data_dir>` for the task `create-data` is `~/tmp/<pkg>/test/src/org/bioconductor/rservicesJms/worker/Data`.

The argument `action` in this ant task corresponds to R function `load` and `data` respectively. If the R objects is provided by the package, you can use `action=data` and provide the object name as argument `data.dir`. The

`action=load` is more useful for loading your own data files or for loading multiple files.

The argument `data.dir` has different meanings on different `action` types. When `action` is `load`, `data.dir` is the path for both the input R data files and the output Java data files. Both absolete and relative path will work. But please make sure all the files in `data.dir` are R data files when you invoke the ant task. When `action` is `data`, `data.dir` is the path for the output Java data file. The argument `data.name` is only used when `action` is `data` and it has to be a R object name, not a R data file name.

3.2 Writing test code – methods

The file

`test/src/org/bioconductor/rserviceJms/services/<pkg>.java`

contains a template for writing test methods. The methods in this class arrange for input parameters to be provided by the developer, and for the corresponding R function to be invoked. The developer is free to implement tests on the return value; the default is to compare the return value with an expected value provided by the developer.

3.3 Running tests

Tests require (1) a running activemq (2) a 'worker' to perform calculations and (3) the Java program to run the tests. The strategy (to be refined) is:

1. Open a terminal window and start activemq

   ```
   cd $JMS_HOME
   bin/activemq
   ```

   (alternatives are in the activemq documentation.)

2. Open another teriminal window, compile the test and package source code, and start the worker:

   ```
   cd ~/tmp/<pkg>
   ant precompile start-worker
   ```

   Several files should be compiled, and the worker should start. The ant task will remain active.

3. Finally, open a third teriminal window and run the test program:

   ```
   cd ~/tmp/<pkg>
   ant local-test
   ```

The test files will be compiled and and executed.

As the test program executes, any output directed toward stderr in R (warnings or errors) will appear in the 'worker' window. Java-based errors (e.g., failed unit tests or explicit print statements) in the test code are echoed in the local-test console, or printed in the test output directory, `test/output`.

4 Creating web services from Java templates

The Java code you have now is a standard Java application. Converting it into a web service application allows your functions to be accessed remotely in a platform and implementation indenpendant way. This process is enabled by Apache Axis , a java platform for creating and deploying web services applications. Please make sure Apache Axis is correctly installed and deployed. If you have no existing web server, use Apache Tomcat as a starting point. Please also specify related properties in `~/tmp/<pkg>/RWebServicesEnv.properties`

4.1 Creating web services

1. Create WSDL from Java code and Java templates from WSDL

   ```
   cd ~/tmp/<pkg>
   ant gen-wsdl
   ```

   The outputs in `~/tmp/<pkg>` are:

   ```
   wsdl/*.wsdl
   wsdl/org/bioconductor/packages/*/*.java
   wsdl/org/bioconductor/rservicesJms/services/*/*
   ```

   The file `*.wsdl` is written in WSDL, the Web Service Description Language. It specifies the type information of your functions, and defines all related data types. It is the agreement between the web service server and client for service invocations. The file is generated by a tool called Java2WSDL from Axis by extracting information from your Java codes. Advanced users can customize the WSDL style via properties `wsdl.style` and `wsdl.use` in `~/tmp/<pkg>/RWebServicesTuning.properties`. The default is `Document/literal wrapped`. More information about WSDL style is available.

   All other Java files in directory `wsdl` are generated by a tool called WSDL2Java from Axis by extracting information from the WSDL file. `wsdl/org/bioconductor/rservicesJms/servi` contains server binding skeletons, client binding stubs and a template for test. The stubs and skeletons handle all the low-level details of the remote method invocation. They allow seemless interactions between your Java application, Axis and web service clients. `wsdl/org/bioconductor/packages/*/*.java` are Java implementations for the data type definitions in WSDL.

2. Creating web service server and web service client

   The outputs from WSDL2Java need to be connected with your Java codes.

   ```
   cd ~/tmp/<pkg>
   ant mkserver
   ant mkclient
   ```

   Two directories are created: `server` and `client`, to hold all data for the web service server and client respectively. The client is only for testing pupose. Any users of your web service can create a client from the WSDL file, by using any tool or any programming language.

The ant tasks gen-wsdl, mkserver and mkclient can also be invoked in one composite task:

```
cd ~/tmp/<pkg>
ant ws
```

4.2 Deploying the web service to Axis

To deploy the service:

```
cd ~/tmp/<pkg>
ant deploy-serv
```

If it fails, check Tomcat log files for error messages. Please also access your Axis instance from browser, and view the list of deployed web services. Sometimes the service does not appear on the list even if the above ant call returns no error information. Try the ant call again. You may also want to restart Tomcat server after deploying the service. The deployment step copies `wsdl/org/bioconductor/rservicesJms/services/*/deploy.wsdd` to the file `<AXIS_HOME>/WEB-INF/server-config.wsdd`.

Always remember to undeploy the service afterwards:

```
cd ~/tmp/<pkg>
ant undeploy-serv
```

4.3 Testing the web service

Add test code to

```
client/*/src/org/bioconductor/rservicesJms/services/*/*TestCase.java
```

Make sure activemq, the 'worker', and Tomcat are all running, and then perform tests:

```
cd ~/tmp/<pkg>
ant web-test
```

Test output is collated in `client/test_output`.

5 Adding Java code to R packages for redistribution

After R methods have been exposed and working tests developed, a next (and optional) step is to add the Java code to the original R package. In this way, the combined R and Java code can be redistributed for others to use or deploy as web services.

The approach is to add Java files to the directory `<pkg>inst/rservices`. The commands

```
ant map-package unpack-package -Dpkg=<pkg>
```

will then create an RWebServices skeleton as outlined for `map-package`, and then copy the files in the `inst/rservices` folder into their corresponding location in the skeleton. The typical contents of `inst/rservices` might be Java source files and perhaps data instances used for implementing tests or simple clients.

6 Alternative deployments: caGrid services

RWebServices packages can be used as traditional web services, or integrated into other projects. One example of the latter involves caBIG and caGrid. caBIG is an effort by the US National Cancer Institute to develop standardized software that uses strongly typed data. caGrid builds on this foundation to offer analytic and data services in a grid-based computing environment built on top of the Globus toolkit.

Here is how one might proceed to create a caGrid analytic service based on an RWebServices-enabled package; the assumption is that caSurvey contains functions with `typeInfo` applied. caSurvey has been built with `R CMD build -clean caSurvey`. One can then

```
tar xzf caSurvey_1.0.tar.gz
R CMD INSTALL --clean caSurvey
echo "library(RWebServices);unpackAntScript('caSurveyImpl')" | \
    R --vanilla
cd caSurveyImpl
ant map-package -Dpkg=caSurvey
```

To start the project. Just as described above, this creates src/ and test/ directories. the test directories are meant to be populated with unit tests to ensure that data are being translated between R and Java correctly (RWorkerDataTest.java) and that the service is invoked correctly (caSurveyTest.java). The worker tests require RWebServices, SJava, and caSurvey to work correctly; the service tests also require activeMQ and a worker to be working correectly. The tests constructed and run as described above.

You can go on to create and deploy a web service (ant ws deploy-serv), but for the workflow we want the next step is to use caGrid and the introduce tool

to create a grid service. We will forward grid service requests to the caSurvey application created by RWebServices' map-package.

Creating a caGrid analytic service is document in this best practices document. Think of application produced by `map-package` as a 'silver level' application (chapter 4), with the goal being to reach 'gold level' (chapter 5). The basic steps involved are

1. Create xsd from the Java data beans produced by RWebServices.
2. Create a caGrid / introduce 'project' based on the xsd and services to be exposed;
3. Add relevant components from the RWebServices project to the caGrid / introduce project.
4. Translate grid service requests to requests handled by the RWebServices project.

The first two steps are necessary when brining any Java project to caGrid, and are described in the caGrid best practices document.

Components of the RWebServices project need to be added to the `lib` directory of the caGrid project. These are:

1. A jar file of compiled classes, e.g.,

   ```
   ant precompile
   jar -cf caSurvey.jar -C bin .
   ```

2. `rservices.jar` from RWebServices, and `activemq-core-4.02.jar` and `geronimo-jms` from activeMQ.

The best practices document suggests that caGrid services use `<service>Impl` to wrap the underlying business logic. For us, this means

1. Import data packaages and the service provider, e.g.,

   ```
   import org.bioconductor.packages.caSurvey.*;
   import org.bioconductor.rserviceJms.services.caSuvery.caSurvey;
   ```

2. Create a persistent service when the grid service is initialized, e.g.,

   ```
   public class CaSurveyImpl extends CaSurveyImplBase {
       private caSurvey caService = null;
       public CaSurveyImpl() throws RemoteException {
           super();
           // Start our service; the service has a lifetime
           // equal to that of this instance.
           try {
               // logs/catalina.out
   ```

```
            System.out.println("Starting caSurvey");
            caService = new caSurvey();
        } catch (Exception ex) {
            throw new RemoteException(ex.getMessage());
        }
        System.out.println("Start caSurvey successful");
    }
    ...
```

3. Forward service requests. The `<survey>Impl` class contains methods. Each method represents a grid service. We map each to a `caSurvey` service, perhaps using `get` methods to access the grid data types. Generally:

```
...
public <caGrid type> <caGrid service>(<caGrid types>) {
    // map input types, i.e., create <caSurvey type>
    // from <caGrid type>
    <caSurvey type> var =
        new <caSurvey type>(<caGrid type>);

    // invoke service
    <caSurvey result> = null;
    try {
        <caSurvey result> =
            caService.<caSurvey method>(<caSurvey types>);
    } catch (RemoteException ex) {
        // maybe log?
        throw (ex);
    }

    // map from <caSurvey result> to <caGrid result>
    return(<caGrid result>)
}
...
```

7 More information

The vignette "Installing and testing RWebServices and enabled packages" provides guidance on package and software installation.

Additional vignettes contain thoughts and 'lessons learned' from this project, and are not essential reading.