Basic API

ReportMill's API is remarkably simple and easy to use. This document starts with a brief overview, then covers all the common API in detail.

Table of Contents

Basic API

Overview

There's only four lines of API that most ReportMill developers will ever use. They use only two different ReportMill objects and both can be found in the com.reportmill.base package. So the first thing you do for any ReportMill code is this:

```
import com.reportmill.base.*;
```

1. Generate XML

You only need to do this once, but you usually want an XML description of your data in order to design a template in the designer. This is optional, but it gives you the nice drag and drop of keys and a design "preview":

```
new RMXMLWriter().writeObject(myJavaDataset, "C:\Temp\Dataset.xml");
```

2. Load Template

ReportMill can load a template from almost any source: a File object, a String path, an InputStream, a URL, a byte array, etc. Most people just use a String path, like "C:\Documents\Sales.rpt", or an InputStream from the Class.getResource() method, but notice that the constructor takes a Java.lang.Object arg:

```
RMDocument template = new RMDocument(aSource);
```

3. Generate Report

Again, ReportMill can generate a report from almost any Java dataset, be it EJBs, custom Java classes, Java Collections classes, JDBC ResultSets or any mixture of the above. There is no interfaces to implement, no feeder API to call, no "DataSource" object to subclass, ReportMill handles databinding automatically using Java Reflection (notice again that this method takes an Java.lang.Object arg):

```
RMDocument report = template.generateReport(myJavaDataset);
```

4. Get Bytes PDF (or HTML, Flash, Excel, JPG, etc.)

Once you have a report, you simply ask for getBytesXXX(), where XXX is any of the supported ReportMill types: PDF, Flash, HTML, Excel, CSV, JPG, etc. Once you have the byte array, you do anything you want with it: write it to file, append it to an HTTP response, insert it to a DB table, send it via Java Mail, pass it over a socket, etc.

```
byte pdf[] = report.getBytesPDF();
```

You can also just write it out:

```
report.write("MyReport.pdf");
```

Basic API

Generate XML from a Java Dataset

The first thing most developers want ReportMill to do for them is create a sample XML file of their dataset, to facilitate template design. ReportMill has a powerful and simple API for generating an XML dataset from any Java object or List of objects:

```
new RMXMLWriter().writeObject(myJavaDataset, "C:\Temp\MyDataset.xml");
```

This call is exactly equivalent to the generateReport method (covered later), except that it includes a "datasetPath" parameter specifying where the resulting dataset is saved. Most developers will pass a simple List to this call.

Degree of Separation

The API above only traverses your object graph three relationships deep by default. This is because some Object/Relational mapping libraries can automatically load an entire database when the Object graph is heavily connected. If the default "degree of separation" of three is not enough for your needs, you can provide a third 'int' parameter to increase this:

```
new RMXMLWriter().writeObject(myObject, "C:\Temp\MyData.xml", 5);
```

Ignore Member / Ignore Class

Your objects may at some point contain a relationship that is of no use in the dataset - or perhaps even dangerous (if you have a public method, like "commitToDB()"). In this case, you can tell RMXMLWriter to ignore that particular field by calling the ignoreMember() method:

```
RMXMLWriter xml = new RMXMLWriter();
xml.ignoreMember("commitToDB");
xml.writeObject(myObject);
```

Debugging/Support

Generating a dataset file is also a great way to provide ReportMill with a reproducible case of a problem. If a specific template/dataset is giving you trouble, generate a dataset and send it, along with your template, to support@reportmill.com for analysis. Support for ReportMill is provided on a fee basis (templates/datasets received without a support contract will be treated as bug reports and may not receive an immediate fix or work-around).

Working without an XML Dataset

It's important to note that the XML dataset is just a convenience during template design. You can add tables using the "Add Table" button at the top of an open document. Keys can be typed directly into table row columns. Even groupings and sortings can be typed right in, using the pull-down menus associated with each list in the inspector.

Basic API

Anatomy of a Dataset

The last section discussed generating an XML description of your Java dataset and the next sections describe turning any arbitrary Java dataset into reports, so it may be worthwhile at this point to explain what ReportMill expects from a Java dataset.

ReportMill expects its Java dataset to take one of several forms: a List of EJBs, a List of custom Java classes, a hierarchy of Java collections, a JDBC ResultSet, or any combination of the above.

Java Reflection

The most common (and the most useful) form for the objects to take is as EJBs or custom Java classes. If your objects fall into this common category, ReportMill uses the features of the Java.reflect package to query the objects for public methods and fields whose names match the text of the keys found in a given ReportMill template.

As you can imagine, this is particularly powerful because you not only get immediate access to all of your existing custom business logic, but you also have the ability to code additional custom methods for use in your reports (which you may also eventually reuse in your web and desktop applications).

Java Collections

Another common form for your Java dataset to take is Java Collections - the most useful of which is Java.util.Map. That interface defines the get(String) method. If your objects implement the Map interface, ReportMill will try to harvest named values from template keys using the get() method.

At this point it might be interesting to note that even if your dataset is originally in the form of a List of EJBs or custom Java classes, you can add them to a new HashMap instance to make other miscellaneous data available to the report:

```
Map myJavaDataset = new HashMap();
myJavaDataset.put("objects", myOriginalListOfEJBs);
myJavaDataset.put("userName", "Fred");
myJavaDataset.put("someOtherKey", "Some Other Value");
```

JDBC ResultSet

Sometimes you may simply want to execute a quick JDBC fetch and generate a report from that:

```
String url = "jdbc:mydbdriver://mydb.domain.com/test";
String user = "whoever", password = "whatever";
Connection db = DriverManager.getConnection(url, user, password);
Statement st = db.createStatement();
ResultSet resultSet = st.executeQuery("select * from MY_TABLE");
```

Basic API

Anatomy of a Dataset (continued)

RMKey.ValueForKey Interface

As a simple option to the Map interface, ReportMill offers the RMKey.ValueForKey interface. This interface simply has the method "public String valueForKey(String aName)". You can implement this interface if your Java objects don't fall into any of the categories previously mentioned.

Raw XML

A final (and rare) option is to send ReportMill raw XML. We highly recommend that you avoid this option if you have access to the original objects (to avoid the overhead of the unnecessary conversion to XML and back). However, if all you have is XML, you can get ReportMill to convert this to Java collections with the following line of code:

```
Map myMap = new RMXMLReader().readObject(aSource);
```

Additionally, if your XML contains any ambiguous types (like an attribute that contains only digits, but you still want to treat as a String), you can configure the Schema by clicking the XML icon in the lower right hand corner of the editor window (after you drag the XML into an open template). This will present the DataSource inspector, which will let you select the Entity and Attribute containing the ambiguous type (where-upon you can set it explicitly with the Pop-up menu).

If you've explicitly set any types in this manner, you can get access to this schema like this:

```
RMDocument template = new RMDocument(aSource);
RMEntity schema = template.getDataSourceSchema();
```

Then you can load your XML with that schema:

```
Map myMap = new RMXMLReader().readObject(aSource, schema);
```

Basic API

A Composite Dataset

The most common use of ReportMill is to turn a simple List of objects into a report. However, ReportMill can generate arbitrarily complex reports based on any number of Lists and/or with sophisticated object hierarchies and object graphs. Such reports usually utilize multi-page templates and/or table groups. Each table (or graph, crosstab, etc.) in these templates has a "List Key", which they evaluate against the top level generateReport() dataset to get their List.

The simplest example of providing multiple Lists of data to ReportMill is just to add them to a Map:

```
Map map = new HashMap();
map.put("Customers", myCustomerList);
map.put("Invoices", myInvoiceList);
```

Then you can call both RMXMLWriter.writeObject() and RMDocument.generateReport() with the new top-level map instead of either of the individual Lists. You can also use this mechanism to provide additional "meta-data" for a report, perhaps to be used as keys in a cover page or page headers/footers. You could do this simply by adding a few quick keys to the same map:

```
map.put("UserName", System.getProperty("user.name"));
map.put("Department", "Human Resources");
map.put("CurrentWeather", "Partly Cloudy");
```

With that, you can now use the keys @UserName@, @Department@ and @CurrentWeather@ anywhere in your report.

Your Composite Dataset may already exist

You app or object model may already contain a class that can act as a composite dataset. For instance, if you have a SessionManager that manages both a getOpenCustomers() and a getOpenInvoices() list, you might simply end up passing the SessionManager object to all your ReportMill calls.

Another good example might be an Invoice object that contains basic attributes, such as getCustomerName() and getCustomerAddress(), but also a List attribute like getLineItems().

Composite Datasets may be arbitrarily complex

ReportMill provides the greatest flexibility in working with your Java datasets. Although you will commonly provide ReportMill with a simple List (of EJBs, custom Java classes, or Java Collections classes) or just a JDBC ResultSet, you can always opt to pass an Object hierarchy containing any of the above.

Basic API (Continued)

Generate Report Overview

ReportMill is perhaps the easiest tool you will use to create your java application. You simply create a template and call three lines of API to generate a report and write a PDF file (from the package com.reportmill.foundation):

```
RMDocument template = new RMDocument("/TemplatePath/Template.rpt");
RMDocument report = template.generateReport(myObjects);
report.writePDF("/MyDocRoot/MyReport.pdf");
```

Alternatively, you can generate a byte array containing the PDF (perhaps to return in an HTTP response):

```
byte pdf[] = report.getBytesPDF();
```

Flash and CSV are just as easy, using the equivalent write and byte methods (for flash and CSV you want to generate the report with the generateReport method that includes the paginate boolean arg set to false, in order to get a single page, scrollable report).

```
RMDocument template = new RMDocument("/TemplatePath/Template.rpt");
RMDocument report = template.generateReport(myObjects, false);
report.writeFlash("/MyDocRoot/MyReport.swf");
byte flash[] = report.getBytesFlash();
report.writeCSV("/MyDocRoot/MyReport.csv");
byte csv[] = report.getBytesCSV();
```

ReportMill gets data from your objects dynamically, using the Java Reflection API. If your template has the key "@revenue@", ReportMill will query your object for a public method or instance variable with the same name. The objects that you provide in this API are used to evaluate the keys in your template. No other wiring is needed!

RMDocument Constructors

```
public RMDocument(String apath);
public RMDocument(InputStream anInputStream)
```

The RMDocument constructor takes a String which represents the absolute path to the template. Your app server may have API to find the absolute path of an application resource or you may choose to keep your templates in a hard coded directory.

There is also an InputStream version, which allows you to have templates as jar resources and pass the input stream to ReportMill for initialization.

Basic API (Continued)

The RMDocument.generateReport() Method

```
public RMDocument generateReport(Object object);
public RMDocument generateReport(Object object, boolean paginate);
public RMDocument generateReport(Object objects, Object userInfo);
public RMDocument generateReport(Object objects, Object userInfo, boolean paginate);
```

The RMDocument generateReport method returns a new RMDocument, which is a filled version of your template from the object(s) provided. The "paginate" parameter is used to tell ReportMill whether you want a multipage document (usually for PDF) or a single-page, scrollable document (usually for Flash and CSV).

The Object(s) Parameter

The "Object(s)" parameter is used to provide the root object for the report. This can be a master "Map" style object, such as a custom Java "Invoice" object (perhaps encapsulating a List of line-items), or it can be an actual Java.util.List object, such as a list of customers. Most reports will have a "List Key" element (tables, labels, graphs). The data for this is either provided directly as the List or as a relation from a master object (see the "List Key" field in the Table Inspector). In this case, each provided object usually corresponds to a row in a table or a single bar/wedge in a graph.

If your template has more than one table or graph or a TableGroup (see the related Advanced Topic), "Object(s)" should be a "Map" style object with several List elements (Invoice->payments, Invoice->Line Items). You can use an actual Java.util.Map object if you need to fill multiple List Key elements from independent lists (in this case the "List Key" fields in the Table Inspector should match the keys in your Map).

If your template is a simple form (perhaps with a PDF backdrop), you data will just be a "Map" style object (perhaps with no List Keys at all). You can even pass a simple Java.util.Map object, with keys corresponding to the keys in the template.

If you want to provide miscellaneous information with a mostly-unrelated List of objects, you can do this by allocating a simple Map to bind the data together.

```
Map map = new HashMap();
map.put("objects", myList);
map.put("userName", System.getProperty("user.name"));
report = template.generateReport(map);
```

List objects provided with the "objects" key will become the default set of objects for use with any List Key element for which the List Key is not set.

Basic API (Continued)

PDF Generation

The most reliable way to return a PDF file to the client is to write the file to your web server's document root and return a reference to the file in your HTML response. Here's three lines of code that generates a report and writes it to file:

```
RMDocument template = new RMDocument("/TemplatePath/Template.rpt");
RMDocument report = template.generateReport(myObjects, myUserInfo, true);
report.writePDF("/MyDocRoot/MyReportRoot/MyReport.pdf");
```

Returning a PDF response

In the context of a Sevlet, the concept is the same, except that you ask for "pdfBytes()":

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
    // Handle "PDF Report" submit button
    if(request.getParameter("PDF") != null) {

        // Get objects and generate report
        Object objects = request.getSession().getAttribute("data");
        RMDocument report = template().generateReport(objects);

        // Generate PDF bytes and load into response
        byte bytes[] = report.getBytesPDF();
        response.setContentType("application/pdf");
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }
}
```

IE POST Bug with PDF

Be aware that returning a PDF response can be somewhat more involved in the context of an HTTP POST (as opposed to GET) call when using Internet Exporer (see Microsoft knowledge base article Q160013 and Q190052). We've found it's usually necessary to detect when the browser is IE and return a frameset or redirect which itself has a dynamic URL request back to the app server for the PDF (which is cached and returned with the second request). There are quite a few Google hits on this problem which might be useful.

Basic API (Continued)

HTML Generation

HTML generation may be slightly more involved than PDF generation, since it may contain image references. If you are absolutely sure your report won't contain images or if you are writing the report to file, you can generate reports as expected:

```
  template = new RMDocument("MyTemplate.rpt");
  report = template.generateReport(myObjects, false); // false - don't paginate
  byte html[] = report.getBytesHTML(); // or report.writeHTML("MyReport.html");
```

However, if your report is likely to reference an image and you need access to both the html bytes and the image bytes, you use use the htmlMap(imageRoot) method:

```
  template = new RMDocument("MyTemplate.rpt");
  report = template.generateReport(myObjects, false); // false - don't paginate
  String imageRoot = "images/"; // This parameter is optional
  Map map = report.getMapHTML(imageRoot);
  byte html[] = (byte[])map.get("htmlBytes");
```

All other map entries will be of the form <image_root>img0.gif, <image_root>img1.jpg, etc. The values will by byte arrays. Here's a sample Servlet doGet() method:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
    // Handle "HTML Report" submit button
    if(request.getParameter("HTML") != null) {

        // Get objects and generate report
        Object objects = request.getSession().getAttribute("data");
        RMDocument report = template().generateReport(objects, false);

        // Generate HTML Map (htmlBytes + resources) and add "Resources" map to session
        String imageRoot = "./Movies?RES=";
        Map map = report.getMapHTML(imageRoot);
        request.getSession().setAttribute("Resources", map);

        // Get htmlBytes and load into response
        byte bytes[] = (byte[])map.get("htmlBytes"); map.remove("htmlBytes");
        response.setContentType("text/html");
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }

    // Handle Resource request
    else if(request.getParameter("RES")!=null) {

        // Get resources map and imageRoot
        Map map = (Map)request.getSession().getAttribute("Resources");
        String imageRoot = "./Movies?RES=";

        // Get resourceKey and resource bytes and load into response
        String resourceKey = imageRoot + (String)request.getParameter("RES");
        byte bytes[] = (byte[])map.get(resourceKey); map.remove(resourceKey);
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }
}
```

Basic API (Continued)

Flash Generation

Flash generation is also straight forward:

```
  template = new RMDocument("MyTemplate.rpt");
  report = template.generateReport(myObjects, false); // false - don't paginate
  reporte.writeFlash("MyReport.swf");
```

Here is a method to return a Flash file from a servlet (it uses an RMHTMLWriter.htmlBytes(aSource) method to get an appropriate HTML-Flash container file):

```
import com.reportmill.base.*;
import com.reportmill.output.*;

public void doGet(HttpServletRequest request, HttpServletResponse response)
{
    // Handle "Flash Report" submit button
    if(request.getParameter("FLASH") != null) {

        // Get objects
        Object objects = request.getSession().getAttribute("data");

        // Generate report
        RMDocument report = getTemplate().generateReport(objects, false);

        // Get Flash bytes and load into a session "Resources" Map
        byte flashBytes[] = report.getBytesFlash();

        // Declare image root
        String imageRoot = "./Movies?RES=FlashReport.swf";

        // Create map for flash bytes resource
        Map map = RMMapUtils.newMap(imageRoot, flashBytes);

        // Add map to session
        request.getSession().setAttribute("Resources", map);

        // Generate container HTML file
        byte bytes[] = RMHTMLWriter.htmlBytes(report, imageRoot, "Report Title");

        // Load container HTML into response
        response.setContentType("text/html");
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }

    // Handle Resource request
    else if(request.getParameter("RES")!=null) {

        // Get resources map and imageRoot
        Map map = (Map)request.getSession().getAttribute("Resources");
        String imageRoot = "./Movies?RES=";

        // Get resourceKey and resource bytes and load into response
        String resourceKey = imageRoot + (String)request.getParameter("RES");
        byte bytes[] = (byte[])map.get(resourceKey); map.remove(resourceKey);
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }
}
```

Basic API (Continued)

CSV Generation

CSV generation is also straight forward:

```
  template = new RMDocument("MyTemplate.rpt");
  report = template.generateReport(myObjects, false); // false - don't paginate
  reporte.writeCSV("MyReport.swf");
```

Here is a method to return a CSV file from a servlet:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
    // Handle "CSV Report" submit button
    if(request.getParameter("CSV") != null) {

        // Get objects and generate report
        Object objects = request.getSession().getAttribute("data");
        RMDocument report = template().generateReport(objects, false);

        // Generate CSV bytes and load into response
        byte bytes[] = report.getBytesCSV();
        response.setContentType("application/vnd.ms-excel");
        response.setHeader("Content-Disposition", "attachment;filename=Report.csv");
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }
}
```

Basic API (Continued)

JPEG Generation

JPEG generation is also straight forward:

```
  template = new RMDocument("MyTemplate.rpt");
  report = template.generateReport(myObjects, false); // false - don't paginate
  reporte.writeJPG("MyReport.jpg");
```

Here is a method to return a CSV file from a servlet:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
    // Handle "CSV Report" submit button
    if(request.getParameter("CSV") != null) {

        // Get objects and generate report
        Object objects = request.getSession().getAttribute("data");
        RMDocument report = template().generateReport(objects, false);

        // Generate CSV bytes and load into response
        byte bytes[] = report.getBytesJEPG();
        response.setContentType("image/jpeg");
        response.setContentLength(bytes.length);
        response.getOutputStream().write(bytes);
        response.getOutputStream().close();
    }
}
```

Basic API (Continued)

Concatenating Multiple Reports

Several reports can be generated separately, then concatenated together, with a simple call to RMDocument's addPages API. Then add the pages together and generate the PDF, Flash or CSV data. Page numbering will then be global to the whole document, unless you explicitly call 'resolvePageReferences()' on one of the component documents.

```
import com.reportmill.base.*;

public void generateReport()
{
  RMDocument template1 = new RMDocument("/TemplatePath/Template1.rpt");
  RMDocument template2 = new RMDocument("/TemplatePath/Template2.rpt");
  RMDocument report1 = template1.generateReport(myObjects1, myUserInfo1);
  RMDocument report2 = template2.generateReport(myObjects2, myUserInfo2);
  report1.addPages(report2);
  report1.writePDF("/MyDocRoot/MyReport.pdf");
}
```

Basic API (Continued)

Object Binding - Your own API!

Perhaps the most powerful API in ReportMill is the API of your own objects. ReportMill is an "Object Reporting System". While all of our customers program with Java objects, it's sometimes not immediately obvious how easy it is for a tool like ReportMill to talk directly to Java objects with so little wiring.

Custom Objects and the Java Reflection API

Java has a simple API called the reflection API (java.lang.reflect package) that lets ReportMill query it for instance variables and methods with a specified name. If you have the key "@customerName@" or "@customerNameUpperCase@" in your template, ReportMill will successfully evaluate that key with an instance of an object whose class looks like this:

```
public Customer extends Object {
    public String customerName = "John Doe";
    public String customerNameUpperCase() {
        return customerName.toUpperCase();
    }
}
```

This example shows how you can immediately put custom business logic to use in your reports (even for something as simple as formatting).

Map

Maps are useful Java collection objects that do nothing but store key/value type data. Many customers use these to specify simple data. Here is a sample that would work fine for a template that had a simple table with just the key "@name@" typed into one of the table column text fields:

```
// Create 2 user objects
Map user1 = new Hashtable(); user1.put("name", "John Doe");
Map user2 = new Hashtable(); user2.put("name", "Sue Smith");

// Add them to user vector
List users = new Vector(); users.add(user1); users.add(user2);

// Generate Report
RMDocument template = new RMDocument("/MyTemplatePath/MyTemplate.rpt");
RMDocument report = template.generateReport(users);
report.writePDF("/MyDocRoot/MyReport.pdf");
```

Basic API (Continued)

Object Binding (Continued): The Power of Preprocessing

Sometimes it can be useful to provide objects to ReportMill that have been generated or manipulated on the fly in a preprocessing step. This can allow you to create a virtual data structure that might not currently exist (and perhaps even can't coexist) in your object model.

For instance, you might want to bind two related Lists of objects that aren't otherwise joined by a relationship. For this you would iterate through your two arrays, creating a Map at each index that contains both objects. Then you access elements with keys like "@list1.someKey@" and "@list2.someOtherKey@".

You might also want to bind an extra instance variable to objects in your array. Again you would encapsulate each object in a dictionary or custom class and add the extra key/value or instance variable/method. That new key/value could be the result of a computation, giving the ability to perform a more sophisticated grouping (of course, that sounds like something to add to your object model, if possible).