Implementing a Numerical Data Access Service

Andrew Cooke[*]

October 2008

Abstract

This paper describes the implementation of a J2EE Web Server that presents numerical data, stored in a database, in various graphical and tabular formats.

It describes the overall architecture of the system, the motivation behind the technology choices, and the integration of the different sub–systems into a coherent, efficient whole.

It also explains a few ideas used in the implementation that may be original and/or useful.

Contents

1 Introduction

Below I give a fairly detailed summary of a single system. It is very easy to find introductory articles that describe single technologies, or high-level views that describe architecture in broad terms, but it is harder to find something in the middle ground: details on how to integrate a range of technologies into a coherent whole.

1.1 System Overview

The system described here presents quality metrics in various ways. The data are quite varied — the metrics describe a complex technical process that has

[*]andrew@acooke.org

global scope — and a lot of work (not described here) went into defining a suitable model.

This server, which presents these data in various ways, is only one part of a larger system that collects and manages the data.

1.2 Requirements

The requirements below guided the architecture and implementation.

1.2.1 Generality

The data model described in 1.1 is very general. A previous system produced high quality presentations (in a single format), but had an architecture that did not exploit this generality and so could not easily be extended to display additional metrics.

In contrast, this system must be able to generate displays for any metric. It must also be able to display new metrics as they are added to the system, without any modification.

1.2.2 Modular Views

The server will present data to a variety of different consumers. It must allow for data exploration (browsing), but also provide access to the data for more detailed analysis with third–party tools (eg. spreadsheets). It should be possible to embed images generated by the system in other presentations (both powerpoints and web sites) and it should also be easy to find related views of the same data (given an image, it should be easy to access a table of the values used to form that image, for example).

These requirements imply that the server is a service — that it should fit easily within a Service-Oriented Architecture — but that it must also appear as a user–friendly, interactive web site. This leads naturally to an approach based on Representational State Transfer (REST).

1.2.3 Flexible Data Source

Uncertainty about the final storage technology means that the system must be able to adapt to different data sources. The current implementation uses an SQL database (MySQL or Oracle), but it should be possible to migrate to other (non-SQL) providers.

This requirement limits the scope of any Object–Relational Mapping (ORM). The technology used to interface to the data source should be limited in scope and relatively easy to replace.

1.2.4 Open, Standards

Finally, it must be easy to extend the system.

For this, a clean architecture is necessary. Exposing that via Spring configuration allows new views to be wired into the existing support with little or no modification of the server infrastructure (ie. lower, non–view layers).

A further, implicit aspect of extensibility is that the system must be easy for other engineers to understand and modify. Existing, open, standard solutions come with broad online support.

1.3 Architecture

In broad terms, the architecture follows the standard model–view–controller (MVC) pattern, which talks to the data source via an abstraction layer (Data Access Object, or DAO).

2 Data Layer

The data layer implementation was driven by:

A The requirement described in 1.2.3: that it must be possible to change the underlying data source.

B The need to support quite complex data exploration. This might include, during iterative development, relationships not explicit in the original data model.

C The need for good performance (caching) and graceful handling of updated data (the addition of new values).

D A desire for a system whose properties can be easily understood.

Note that the system is read–only. It presents, but does not modify, data. In terms of the REST approach (see 1.2.2) it handles only GET requests.

2.1 Object Graphs

B suggests that we will have strongly inter–connected data structures. This could raise issues with **C**, in a traditional (transparent) ORM approach, since it may be difficult to expire/replace some sections of the graph. At the same time, **A** and **D** suggest that a simple, explicit solution might be preferable to a more automated approach.

When the information stored in the database is used to generate objects in Java we create a graph. The objects are nodes; the links between objects (typically implemented as references and accessed by getters and setters) are arcs.

A traditional, implicit ORM approach generates this graph of objects automatically. This can simplify the Java code but makes it difficult to control the caching of individual objects (since links between objects tends to keep them in memory).

Instead, for this server, I separated the nodes and arcs. Each object can be retrieved by key, but does not link to any other object. Instead, the interface to the data source provides keys for related objects. This makes some Java code slightly more complex, but makes it much easier to control caching, support different data sources, and expose relationships that were unexpected in the design of the original data model[1].

One consequence of this approach is that the Java code does not have to traverse graphs of objects in various ways to infer relationships. Instead, the appropriate method in the data layer is called. This is currently an advantage because relationships are easy to express declaratively in SQL. However, it may become a disadvantage when moving to a less flexible non–SQL data source.

[1]I am sure it would be possible, with sufficient work and care, to present the solution described here in a more transparent, implicit manner. My argument is only that the explicit approach is simpler and more than "good enough."

2.2 Caching

The data model supports two kinds of objects. Some objects are measurements. Other objects describe those measurements (for example, they identify the property being measured, the type of measurement, any classification applied to the measurement, etc). In this context we can consider the measurements to be "data" and the descriptive objects to be "metadata".

In addition to the data and metadata objects, we also have information about the relationships between objects (these are separate from the objects themselves — see 2.1).

The system uses three caches, for these three kinds of information (data, metadata, and relationships).

- The first cache, using strong references, keeps all the metadata in memory. These objects are flushed periodically so that they can reflect updates to the database, but are otherwise permanently available.
- The second cache, using weak references, manages information about relationships. As much information as possible is cached, but it can be expired by Java garbage collection.
- The third cache, using a limited amount of memory, with expiry of least recently used data, holds recently accessed data (measurement) values.

As a result the server has efficient access to system metadata without blocking progressive updates as the underlying data set is extended. In addition, there is no restriction on the relationships that can be provided by the data layer — new relationships are easy to add during development. The system can be developed in an iterative, incremental manner, but will always work efficiently.

2.3 IBatis

The iBatis Data Mapper framework is a good match to the approach outlined in 2.1 and 2.2.

The final implementation, built on iBatis, uses three levels of interfaces:

- At the lowest level is a thin wrapper around the iBatis client that adds support for Java Generics. Methods here return sets of objects given a named query (the query identifies the SQL to execute).

- At the middle level is a more explicit interface that gives typed access to the various objects and relationships. At this level the relationships are expressed in terms of keys.

- The top level (the DAO itself) is similar to the middle level, but presents relationships as objects (it automates the retrieval of instances given the keys).

Support for a different data source would probably implement the middle layer; the lower layer is too specific to iBatis and the upper layer automates work that any data source would need to implement.

Within this implementation, iBatis provides the following benefits:

- Mapping from SQL to Java objects is easy to configure and modify.

- SQL and mapping information is kept separate from the Java code.

- The caching outlined in 2.2 is easy to configure.

- It is easy to adapt the SQL to the underlying database engine.

3 Controller

The system's core follows the usual MVC pattern, using Spring MVC.

Almost all requests are handled by a single controller, which delegates to a handler (or sub–controller). The handler, selected from a lookup table via four facets (described below), provides the command object and view.

3.1 Facets

Exploring some typical use cases gave a set of four largely orthogonal facets that a user might specify in a request. These are:

Content Type may be inferred from the HTTP request or specified as an additional parameter. It is the main factor in determining which view to use.

Subject describes the kind of data that will be displayed. It may be a simple measurement, an average, etc. This is the main factor in determining what data are retrieved from the data layer (via the command object — see 4.1).

Style describes how the data are displayed. It may be in a table, as a line plot, etc. This typically correlates with Content Type, but is particularly useful in selecting the view when the result will be displayed as HTML (since that Content Type supports both tables and graphs).

Grouping is related to a detail of the data model — grouped data are handled in a distinct way. I will not describe this further here.

By expressing facets within the REST URL we support, in an intuitive, simple way, the selection of different services that display related data. For example, changing the style in a URL from line plot to table will give the data that were displayed in the graph. This URL rewriting can be done by an advanced user, but is also the basis for generating related pages during exploratory data analysis (5.4).

In addition, of course, the user must identify the data to display. But those details do not alter the choice of command object or view. This is a consequence of the very general nature of the data model, as mentioned in 1.2.1.

3.2 Spring

Not all combinations of facets are supported. For example: graphs are not displayed via CSV and XML content types; tables of data are not displayed via PNG images.

Supported combinations of facets, and the associated models and views, are configured via handler objects. The controller searches through the handlers until it finds one that supports the facets inferred from the URL.

Exposing the controller via Spring XML configuration files allows the system to be easily configured. The configuration also serves as a form of documentation. However, native Spring XML configuration is rather verbose, which obscures the relevant details. The use of a custom namespace handler allows a more concise format:

```
<kpi:controller>
  <kpi:handler contentType="PNG" style="BARS"
               subject="VALUES" group="false"
               command="VALUES" view="pngBarChart">
    <ref bean="entityTitles"/>
  </kpi:handler>
  <kpi:handler contentType="XML" style="TABLE"
               subject="AVERAGE" group="false"
               command="AVERAGE" view="xmlTable">
    <ref bean="valuesDates"/>
    <ref bean="valuesTable"/>
  </kpi:handler>
  ...
</kpi:controller>
```

The fragment of XML configuration above shows how each handler associates a model (the command object — see 4.1) and view with a set of facets.

The lists of beans within each handler are “presentations”. These extend the model in various ways and are explained in 4.2.

4 Model

The model is a simple Map from names (Strings) to arbitrary Objects. It is provided to the view via the Spring MVC API.

Construction of the model is a three step process:

- Simple values (the facets described in 3.1 and additional URL parameters like start and end dates) are extracted from the HTTP request. This step may include calling the data layer to convert from keys to object instances.
- Data (measurements) may be requested from the data layer.
- Presentations modify and extend the model.

4.1 Command Objects

Command objects handle the first two steps above. Although Spring MVC includes support for command objects this system uses custom code that is closely integrated with the URL rewriting described in 5.4.

The command object pattern is the transparent binding of attributes in an object to values inferred from the HTTP request. Typically, support code instantiates an instance of the appropriate Java bean and uses reflection to populate its attributes with HTTP parameters.

In the implementation used here annotations identify instance variables. These are bound to values from the request. In addition, sufficient information is stored in the model to reconstruct the URL, possibly in a modified form (see 5.4).

So a request is handled with the following steps:

- The controller selects a handler based on the incoming request.
- The handler specified a command object class, which the controller uses to instantiate an instance.
- The command object support code “magically” populates the command object’s instance variables (which are annotated appropriately).
- A method is invoked on the command object (which implements a standard interface) that does the work necessary to construct the model.

The advantages of using this pattern are that the model construction is separated from the details of exactly how particular values are extracted from the HTTP request.

4.2 Presentations

Once the command object has generated the initial model, it is passed to the presentations. These usually perform simple modifications that are common across many different handlers, such as adding title and axis labels.

Command objects extract data of a particular type while presentations are responsible for more general processing. So presentations and command objects tend to provide orthogonal functionality. Dividing responsibilities in this way simplifies the code while making the system as a whole more flexible.

5 Views

The Spring MVC framework supports a wide variety of views. These are specified in the configuration — there is no need to modify any code when introducing a new view technology.

5.1 Tables

XML and CSV tables are generated by dedicated Java classes that write directly to the servlet response.

The HTML table is generated by a simple JSP page (the table is present in the model and is iterated over by JSTL tags).

In all cases the model data, extracted from the data layer by the command objects, is modified by a presentation object to create List⟨List⟨String⟩⟩ values that are easily handled by the appropriate view. One presentation uses introspection and can be configured to generate data from a variety of classes. In other cases it was simpler to write dedicated presentations.

5.2 Images

Images are generated by Java classes that call the JFreeChart library.

5.3 Embedding

PNG images, as described in 5.2 are returned when the content type is appropriate, but it is also possible to request the same data in an HTML page. In this case a JSP view is called which constructs a page that embeds the PNG image using a ⟨img⟩ element. The URL for the image is identical to that used for the surrounding page, except for the content type facet (3.1). When the HTML page is rendered by the client's browser the server is called again and the PNG image itself is generated.

5.4 URL Rewriting

Changing the content type, as described above, is a simple form of URL rewriting. JSP pages have access to a wide variety of rewritten URLs via functions implemented in the JSP Expression Language (EL). These functions are static, but take as first argument an object that is added to all models and which provides modified versions of the current URL (see 3.1). The static function dispatches the request to this object.

This approach is somewhat unusual — it is more common to expose functionality that depends on the model as custom tags — but the compact expression language syntax gives a cleaner page structure.

5.5 Style

The Yahoo User Interface (YUI) library makes cross–platform CSS easy and simple (including 3 column layouts).

5.6 Ajax

Almost all HTML views are constructed on the server by a few JSP pages. The emphasis on a REST interface and the ability to rewrite URLs by changing facets (3.1, 5.4), enable a rich interface through quite simple HTML.

However, the initial selection of which data to plot is complex and involves such a large number of options that a dynamic AJAX solution is needed.

On the server side a dedicated Java class calculates the children for any particular node in the menu. This is a plain Java object, configured in Spring.

On the client side, the YUI toolkit is used to present the selection as a dynamic tree menu.

The connection between the server's Java code, which calculates the menu structure, and the client side Javascript that presents the data, is handled by Direct Web Remoting (DWR). Only a small change to the Spring configuration on the server is required. DWR then uses introspection to generate a Javascript proxy. The result is a Javascript object on the client that transparently calls the server code.

6 Testing

Unit testing was performed with JUnit. The modular nature of the architecture makes it easy to test most parts of the system.

Integration testing is also performed, using Jetty within JUnit. This is very useful and provides excellent code coverage.

7 Conclusion

7.1 Technologies

The following technologies worked well together:

Spring MVC [1] provides the MVC core.

JSP (using JSTL and EL [2]) simplified generation of "basic" HTML views.

JFreeChart [3] was used to generate images on demand.

YUI [4] provided CSS and Ajax on the client.

DWR [5] exposed server–side Java logic to Javascript objects.

IBatis [6] simplified the data layer and provided caching.

Spring configuration allows extensibility and supports customised syntax for the controller.

7.2 Ideas

The following ideas were helpful in this particular case:

Explicit Object Graphs in the data layer (simplified caching and replaceability) — 2.1.

REST, Facets, and URL Rewriting support both data exploration and use of the system within a broader Service–Oriented Architecture — 3.1.

Presentations provide functionality that complements command objects – 4.2.

EL Functions Dispatch on objects in the model[2]. This gives cleaner JSP code that custom tags – 5.4.

7.3 Example: Image Map

As a final summary I will outline how extra functionality was added in a late development iteration.

At the start of the iteration it was possible to generate an image and embed that image in an HTML page. But it was not possible to click on the image to "drill down" to increased detail. To enable this an image map had to be added to the HTML page.

JFreeChart supports the generation of image maps, so the work required was:

1. Extending the classes that generate the image so that they could also produce an image map (via a separate call). This used the URL rewriter to generate appropriate URLs for various parts of the image.

2. Locating the handler for the HTML page in the controller configuration and changing the command object to match the one defined for the PNG image handler (initially the HTML page command object did not retrieve the data to be plotted; it is now needed to generate the image map).

3. Adding a presentation to the HTML page handler that calls the code written in step 1, adding the image map to the model.

4. Extending the JSP view so that, if an image map is present in the model, it is included in the HTML output.

Note how little of the existing code had to be changed to add this extra functionality. Almost all

[2]So, for example, the model may contain a Foo object which has an instance method doSomething(...) and a static method doSomething(Foo, ...). The static method is added as a custom function, called with the Foo instance as first argument, and invokes the instance method.

the modifications are simple additions or adjustments to the configuration. This demonstrates how the system supports extension with little risk of damaging existing functionality.

Also, because recent measurement data is cached, this extension comes with little cost in response time. The call to generate the PNG image, made as soon as the HTML page is rendered, re–uses the cached data.

8 Acknowledgements

This work was done for ISTI [7], who are nice enough to let me implement whatever I think makes sense (and then let me write about it).

References

[1] Spring Application Framework
http://www.springframework.org/

[2] JSP Expression Language
http://java.sun.com/developer/EJTechTips/2004/tt0126.html

[3] JFreeChart
http://www.jfree.org/jfreechart/

[4] Yahoo! User Interface Library
http://developer.yahoo.net/yui

[5] Direct Web Remoting
http://directwebremoting.org/

[6] IBatis
http://ibatis.apache.org/

[7] ISTI
http://www.isti.com/