Sportwire II: SportPage User Guide

Creating Dynamic Websites with SportPage

teledynamics communications

$Revision: 1.11 $

$Date: 2002/03/13 18:12:33 $

Gary Lawrence Murphy, Teledynamics Communications Inc[1]

Copyright

SportPage is the website presentation component of CBC Sportwire, an archiving and query system for XML documents designed for delivering sports newswire services. The SportPage component is a general purpose Jakarta-Velocity web application template framework, similar to but less complex than Apache Turbine.

The latest edition of this document is available online[2] and as a printable PDF file[3]. For more information on SportPage and the CBC Sportwire news system, please visit the Sportwire home page[4].

[1] http://www.teledyn.com

[2] http://sportwire.sourceforge.net/userguide
[3] http://sportwire.sourceforge.net/userguide.pdf
[4] http://sportwire.sourceforge.net

Table of Contents

prelude

CBC Sportpage is the website presentation component of Sportwire[5] and is primarily designed to deliver the Sportwire news data for the 2002 Winter Olympics website and for the general CBC Sports[6] website.

SportPage was inspired by Apache/Jakarta Turbine[7]. It shares the same Model-View-Controller(MVC) design where website topic areas are defined by the URI, the context can be manipulated based on the URI, and you may create Java classes to be associated to specific URI path points. Unlike Turbine, Sportpage is intended only for public-information websites; there is no concept of a session, no authentication, and, in the current release, database access is via generic JDBC with SQL.

Why another template system?

SportPage is designed to meet the following special requirements:

- SportPage is designed for read-only access websites under extreme web-traffic conditions; the site must accommodate sustained server loads in excess of 40 simultaneous clients, and serve millions of pages per day. The server must also allow for migration to distributed server architectures.
- Must integrate with existing Java servlet infrastructure. The CBC systems use Apache Tomcat with the Apache webserver, running on HP-UX. The News Template system uses javabeans to access news data and formats these through JSP.
- Allow multiple independent deployments on the same server hardware, accommodating independent development teams and virtual hosts. The software must allow multiple instances of itself to be applied to completely independent deployments on the same hardware. Olympics configuration or operation must not conflict with Sports, Sports must not conflict with regional news and so forth.
- Model-View-Controller pattern where the model may be in a database, dynamically generated or retrieved from static files, the controllers are primarily Java objects and may come from any arbitrary (even off-site) sources, and the views are defined in HTML.
- Separation of development into clear casts. Systems administration and deployment must be completely separate from Java development, and both are distinct from web layout, all three distinct from content generation. Journalists do not code HTML, HTML layout artists do not code Java. Java developers do not mess with system init scripts. This is a basic reality of any large webserver project.
- Related to the above Controller definition, SportPage had to allow page designers the freedom to incorporate *any java object* without requiring support from a Java programmer. If some other department is able to provide a basic javabean for access to some data item, incorporating this into the webpages should not involve any java coding and must be entirely contained within the template and configuration files under the designers' control.
- Layout and content changes by design and editorial staff must be implemented on the fly, with no intervention from java programmers or system administrators.
- SportPage must place no constraints on the presentation layer.[1]

[5] http://sportwire.sourceforge.net/
[6] http://cbc.ca/sports/
[7] http://jakarta.apache.org/turbine

[1] the relationship between `$layout`, `$page` and `$index` does place a mild constraint on page templates; because directories cause the page variable to be replaced by the index variable, the master layout template must use at least one `#parse($page)` call, even if that is the only line in the template.

- Human oriented presentation. *All URI strings must be easily spoken by the on-air people*, topic paths must make logical sense in natural language, and no unnatural terms should be visible.
- Guaranteed delivery. As the engine of a website for a National Public Broadcaster, the SportPage server must provide graceful recovery for system errors such as loss of database connections, missing files or parser errors. Any error messages seen by the visitor must be seen as branded message pages and not as infrastructure stack traces or other developer messages.
- The template system must use "topic oriented" screen templates. While overall layout and page-components may be shared, the interpretation of context-dependent information must be decidable through inspecting the extra path information in the URI. For example, in `/olympics/sports/luge` the meaning of `$standings` will point to a different database cache object than it would in hockey.
- Maximum re-use of developed resources. Graphic design and layout talent is expensive: Templates need to use generic methods which can be interpreted according to the path context.
- Basic support for transforming XML objects into HTML.
- Ad-hoc insertion of SQL result sets and XML documents without requiring Java programming. Ideally this should not involve any systems administration involvement.[2]
- SportPage must accommodate a development process where the development and staging servers have no hope of emulating the entire web site. Database access and other live-systems feeds must fail gracefully when unavailable.

While several other portal and web application systems can provide these facilities, most notably Jakarta Turbine[8], no existing package could provide all of these features without carrying other overheads such as authentication and forced sessions support.

Scope of this Document

This paper is intended for website designers and content authors who are using SportPage to present their work. It is not a SportPage developers' Guide. Unfortunately, using SportPage will require some basic knowledge of Java's object-oriented syntax; you need not be able to code in Java, but you need to understand syntax such as `item.attribute`, and understand how objects provide access to the attribute values of the object (it's current state) and provide *methods* which either modify the object state, or retrieve more complex information from the object.

The Map is Not The Territory

SportPage is an open source project. This means a rapid release cycle and an organic architecture that will evolve over time to meet immediate requirements of the sites deploying SportPage servers, and it means aspects of the project such as documentation and unit testing may not be in sync with the actual code. Even the online API documentation[9] may not be 100% current.

If you plan to deploy SportPage, do not depend on the online documents. The most reliable map can be generated by downloading the latest release or CVS snapshot from the project home page and then generating the API documentation with the **Foreign/genapi.sh** script.

[2] While this was the goal, in practice, the use of the configuration file definitions for SQL objects requires a servlet container reset, and this requires system admin interventions. On-the-fly reloading of the configuration is on the ToDo list.
[8] http://jakarta.apache.org/turbine
[9] http://sportwire.sourceforge.net/api

Glossary

I don't care what I said, you know what I meant
--

SportPage should not require an undergraduate computer science degree to deploy the technology. Unfortunately, because of the fuzzy boundary between Velocity and the Java programming language, there are a few techno terms that creep into this document. The following list explains a few of the more esoteric items, for others, it's probably a good idea to ask Google[10].

SportPage Terminology

Veloticy template	SportPage is based on the Velocity template engine[11], a multithreaded template system for merging dynamics information with a static layout. For details on the syntax of Velocity, read the Velocity User Guide at the Apache website.
URI	Universal Resource Indicator, or the path part of a URL. For example, in the URL `http://cbc.ca/olympics/sports/hockey`, the URI is the section after the host name, `/olympics/sports/hockey`. SportPage is fundamentally tied to the URI as defining the page path and *topic*.
Topic	In the context of SportPage, the topic and the URI are the same thing; a web page *path* is computed by removing the webapp path, and the topic by replacing internal path slash characters with periods. In the above definition for URI, the webapp is the `/olympics` portion, the path would become `/sports/hockey` and the equivalent topic would be `sports.hockey`
presentation template	This is a legacy term from the old CBC News system where the "template" meant the form used by journalists to enter news items; the term became widely adopted to mean "CBC-generated news items for the website". In the context of SportPage, the `$weather` and `$news` javabeans are called "template" objects because their data has its origins in the News Template system. Where possible, this document will differential these from Velocity templates by explicit qualifiers.
Page Handler	The SportPage `PageHandler` interface defines an API for java classes mapped to specific URI paths that process all URIs under that path. Page handlers are invoked to do the context setup for these pages and are able to change any context values prior to the template being processed. To be located by the loader, new page handlers must be in the `ca.cbc.sportwire.servlet.pagehandler`.
javabean	In the most general sense, a "Java Bean" is a java object with a null constructor and a standard convention for access to object attributes. For Velocity, a Java bean must have a null constructor, access methods and provide all list and lookup methods as `List` and `Map` classes; third-party Java objects returning other list and map types will need to be wrapped in another class. For an example of a class wrapper, see the `VelocityCalendar` source code.

[10] http://www.google.com
[11] http://jakarta.apache.org/velocity

JDOM	Java Document Object Model is a simplified form of the W3C Document Object Model specification. Don't ask what the latter is, just be happy that access to the former type of objects is so simple. See the example templates (Example 1) in this document for access methods. For more information on JDOM, see the JDOM Project Page[12].

Installing SportPage

SportPage is distributed as a Java Servlet Web-Applications Archive (`sportpage.war`); in most modern servlet containers, this file should be placed in your `webapps` directory and your servlet container restarted.

To upgrade an existing system, the war file can be queried to extract specific components, so you can extract just the `WEB-INF/lib` to update only the SportPage java libraries.

SportPage also depends on several foreign Java libraries. To protect against version conflicts, the specific version of the binary Java jar files are included in the `sportwire/Foreign` package; sources for all of these jar files are freely available on the net or from the Sportwire authors. For a full list of jar files *not* included in the distribution, see the `README` file in the SportPage package.

Basic Configuration

Almost anything is SportPage can be manipulated through the `WEB-INF/sportpage.conf` configuration file. Throughout this user guide you will be referred to this file to make local changes, and because the developers spend so much time with this config file, the comments in that file should be taken as the closest thing you are likely to get for an authoritative reference.

The configuration file is based on the Apache Commons `ExtendedProperties` specification; it is not a standard Java property file but is more similar to an Apache configuration file. Critical differences include

- Non-string items are validated by the loader and rejected if the formats do not conform to the Java specifications. These include dates, integers, and floating point numbers
- Lists composed of strings separated by commas are represented in the SportWire applications as `String[]` (string arrays) and not strings; if you take the `toString()` method of a list variable, you will get only the first element of the list.
- the special token `include` indicates a secondary configuration files to be included at this point. The filename may be absolute or relative to the current directory.

[12] http://jdom.org/

Creating Templates

SportPage templates are implemented in the Jakarta Velocity template system; before you read this section, you should read the Velocity User Guide[13] to better understand the terminology and the Velocity VTL syntax.

SportPage uses Velocity variables to define page components common to most webpage design. By defining an overall layout and the placement of the header, menu, content page and sidebars within it, the website achieves the necessary consistency for good design while preserving and conserving (expensive) designer effort. As will be explained later, SportPage allows the designer to swap in different components depending on the context of the request, for example to give different topic areas context-specific menus, sidebars or content page layout.

Selecting a Template Path

The first important question about templates is “where do I put them?” The factual answer is “SportPage doesn’t care.” When templates are requested, Velocity is told to search the comma-delimited paths listed in the `templates.home` configuration variable. In practice, it is a good idea to keep templates outside of the webserver `DOCUMENT_ROOT` tree, but designers may choose to keep them here so their testing of pages will function correctly loading the templates as static webpages in their HTML editor.

Rule #1: Software Should Serve People

It is important that software serve the people who use it, not the other way around. It is more important to satisfy the process requirements of the website designers than to satisfy server or security ideals and it is best to consider the entire gestalt of the software ecology. Engineering is about solving problems, not imposing new ones.

If your normal Apache webpages will be at `/sites/cbc.ca/docs/olympics`, there’s a trade-off here as to whether or not you want anyone (developers included) to be able to view the raw templates outside of the servlet. One argument goes that you don’t want these pages inside `/sites/cbc.ca/docs` because robots might accidentally index them; I don’t think it matters much.

Caution

If you put the templates outside of the server document tree (e.g. in `/sites/cbc.ca/templates/olympics`) the Tomcat and Apache servers won’t be able to access them. The problem for designers will be that the templates and the images they use will be in different places because the images must be accessible to either tomcat or apache.

Servlets, Apache and Static Content

By default, and this varies from platform to platform, Apache will not allow servlets to serve static HTML content, specifically URI requests ending in `.html`. This is a problem if your templates or your default content (extra path information) ends with that suffix: This is one very strong *technical* reason why all internal links should never include these extensions. Unfortunately, the Apache server does not impose this same rule on static images: To accommodate this, SportPage is equipped to handle `*.gif` and `*.jpg` files found relative to the servlet home

```
#include ("bio/header.html")
```

[13] http://jakarta.apache.org/velocity/user-guide.html

would mean the *template* at `/sites/cbc.ca/olympics/bio/header.vm` whereas

```
<a href="bio/bio.html"><img src="bio/bio.gif"></a>
```

expects to find the *servlet-home* relative file path `/sites/cbc.ca/site/webapps/template/bio/bio.gif` but will create a link that will return the *Apache DOCUMENT_ROOT* relative file `/sites/cbc.ca/site/docs/bio/bio.html`

What works for me is keeping images and templates in different places. Instead of an image url like `/olympics/gfx`, I'd use `/gfx/olympics` so the tomcat is only asked to deliver templates when people ask for `http://cbc.ca/olympics`; all images are specified as `/gfx/...` and no images are specified as `/olympics/....` This avoids a raft of problems, and distributes the load so Apache (or khttpd) handles static content while tomcat does its tomcat stuff.

Second, I would keep the templates outside of the servlet home directory because that lets us upgrade the Olympics software in the easiest and least dangerous way: we just delete it, wiping it clean, and unpack the new kit.

This may be counter-intuitive to those accustomed to using static pages, but I find that it only makes sense to bundle pages with the software when the software and the pages are both done by the same people. It reinforces the division "this is logic, and that is presentation"

Thus, I'd probably ask to put my templates under directories like `/sites/cbc.ca/templates/olympics/bio` and encourage others to use paths like `/sites/cbc.ca/templates/sports/hockey`.

It's largely a personal choice of what is most convenient for everyone involved; SportPage doesn't care. You could, if you want, insist that upgrading software is done by unpacking the new software over top of any previous install (manually running **jar xf olympics.jar**) and keep all your templates under the one and only application root path `/sites/cbc.ca/docs/olympics` so everything is in one place.

Template Structure

In the CBC sites, we copy the same basic webpage component design you see everywhere: Header at the top, menu down the left side, main content covering the rest though sometimes losing it's right margin to a sidebar.

In SportPage, these appear in the configuration file as the first of the `default.*` dot-variables. Dot-variables are so important, they have been badly named; more about them later, but for now, you'll see that the configuration file (`WEB-INF/sportpage.conf`) lists these components as assigned to filename paths such as `/default/layout.vm`.

Note

All page component filenames and directory paths are completely arbitrary. There is no reason why you couldn't assign these layout components to file paths divided by developer (`default.layout=/jones/crusher.vm`) or any other structure; all that matters is that the default set define template filenames, and that those files exist and are readable by the server before the SportPage system is started.

Also note that *only the `default.layout` variable name is required* All the other page component names can be anything you wish, in any language (supported by the file-system) *providing the variable names match the names used in the default layout file*. For example, you could define `default.centerstage=/sponsor/molsons.html` and then place **#parse($centerstage)** into the center of your main content area.

Defining Template Macros

SportPage uses a customized `VelocityEngineTemplate` which allows SportPage to co-exist on servers running other Velocity-based applications. As such, SportPage maintains its own velocity template library; the path to this library is specified in the `vtl.macro.lib` parameter in the `WEB-INF/sportpage.conf` configuration file.

Warning

Velocity macros are not reloaded when the template pages change (they can be, but it's too expensive to be practical). Do not include in-line or library macros until you are certain they work; changes to these macros will require a servlet container restart.

Topics, Contexts and Template Dot-vars

A SportPage application is a collection of topics, each consisting of a collection of subtopics. A primary requirement of SportPage is that these topics be queried by human beings using as close to a natural query language as is possible within the web. To do this, SportPage combines two architecture ideas: A human-oriented query vocabulary and Representational State Transfer (REST). The latter could be simply put as "To the reader, everything is a document" and manifests as doing *everything* over plain ordinary HTTP, and the former involves an avoidance of GET-style appendages (for example, "?x=0xDE&y=2") and extraneous technology artifacts such as file-type identification suffices (for example `.html`, also known as the ubiquitous "*dawt aych tee emm el*").

From Human Need to Machine Query

SportPage URI paths are intended to be easy to say in a natural and unambiguous way while avoiding file-type identification suffices and extraneous characters. For example, the human visitor is interested in "olympics, sports, luge, history" and when the translation from that phrase to the URL is explained once, the visitor has a pretty good chance of guessing the URL to "olympics, sports, hockey, essentials".

A design philosophy of SportPage is to let people do what people do best and let machines do what machines do best. Machines are very good at looking up things, people are not very good at remembering details that are out of context with the subject matter. A further design philosophy is to "humanize" our information systems; *only developers care if a page is sourced from HTML, XML or dynamically generated.* All people care about is getting at the content, in this case, information about sporting events.

This human interface engineering is accomplished through the `default.extensions` list (specified in the configuration file) and a rule that the index of a page is called either `index` or `home`. Given a list of extensions of `.html,.xml`, a URL like `http://poseidon/olympics/sports/bobsleigh` (with or without the trailing slash) will try the following sequence looking for a valid page file

1. `/olympics/sports/bobsleigh/index.html`
2. `/olympics/sports/bobsleigh/bobsleigh.html`
3. `/olympics/sports/bobsleigh/index`

4. `/olympics/sports/bobsleigh.html`
5. give up and insert the 404 page-not-found exception which the template can reference as `$exception`.

The reason for the second rule is because that was what the CBC designers preferred. The reason for the fourth is because some Apache installations will usurp `/olympics/index.html` even though it may allow `/olympics/sports/index.html`... go figure, but I expect this is because of limited pattern matching inside `mod_jserv`. The work around is for the root index HTML-pagelet to be named `index`

A page is a page, an index is an index

A corollary design philosophy of SportPage is a semantic difference between a page within a topic and the index page of that topic. For this reason, SportPage will flag any URI which resolved to a directory and can handle this differently than URI specifications that resolve to a file if an index page exists.

One application of this rule is in the handling of pages using the `DOMFileFactory`. Normally, the extra path information is used to key a specific XML document within the topic area, but if this extra path is null (as in an index page) then the DOM object is undefined; by making the URI resolve to an index page, the application can specify a different content page and/or change the template (in this case the template `$index` is substituted for `$page`).

From Machine Query to Context

A SportPage URI path defines a context and may also define qualifying information about that context. In the most basic case, `/sports/hockey` would define a final context topic (hockey) with its own menu, layout, news headlines &c, while `/sports/hockey/news/story/brandon23` could resolve to a context of `hockey.news`, thereby setting the main content area template to a news-story formatting template, and passing the extra information as the unique identifier to retrieve the specific news item from the database or from a file.

Sportwire decides on the context in two ways, by actions of the page handler, and by context dependent dot-variables.

In the first method, the original URI string is scanned from the end to the beginning looking for a path component that describes a `PageHandler` (described later); if a handler is found, the class is instanced and may alter the context-building process, for example, to take the remainder of the path as a default document or as database key information. This variation in the action taken on the extra path info can only be manipulated by creating page-handler classes.

Example 1. Deciding the SportPage Page-Handler

A request URI is scanned from the end to the beginning looking for a matching Java page-handler class:

```
http://cbc.ca/olympics/bio/snowboarding/deadhead/jim.bo
```

this url gets scanned for `DeadheadPageHandler`, then `SnowboardingPageHandler` and finally settles on `BioPageHandler`. This `PageHandler` is dynamically loaded and given the current page context (a Map of `$varname=object` pairs) which the handler can freely modify before the page starts to render.

The most common use of special page handlers is to restructure the page layout by changing the context variables specifying the sidebars, pages or any other context variables.

The page handler may also add other variables, for example, a `BioPageHandler` might take the `$path` as an XML document and pre-loads it as a globally accessible `JDOM` object that the `/bio` page templates can share.

The following code sample shows how a global `$dom` variable bound to a shared `JDOMFile` object might be used in a template:

```
#set ($root = $dom.rootElement)
#set ($sp = $root.getChild("sport"))
#set ($sport = $sp.getAttribute("name").value)
#set ($athlete = $sp.getChild("athlete"))

#set ($ch = $athlete.getChild("career_highlights"))
#set ($chs= $ch.getChildren("point"))
#foreach ($point in $chs)
   #show($point)
#end
```

In practice, however, this method is not as useful as it seems and it locks the types of content to specific topic areas. A more flexible method is to define dot-variable beans, which can be specified globally or per-topic, and then use these beans to access documents by passing in the `$path` pathinfo. The above example would be modified only slightly, by inserting the following code at the top:

```
#set ($dom = $xml.dom( $path ))
```

The javabean dot-variables `*.bean` lets you map any java beans to variables; in this case, `$xml` has been bound to the `DOMFileFactory` which provide the means to fetch documents from the `$path`. If the javabean uses data caching and/or is a singleton (or stashes itself in the application properties), there is very little overhead in using this method.

Overwriting dot-variables happens when the page handler is loaded, starting from the left-end of the URI, adding each path element one at a time as it scans the configuration file for variables beginning with that same string. For example, given the URI `/olympics/sports/hockey/history` the page handler will assign all `default.*` values to variables of the same base name (`default.lineup.category` becomes `$lineup.category` in the velocity template), then repeat the process for the first level (so `sports.lineup.category` overwrites the value of `$lineup.category`), then adding the second (`sports.hockey.lineup.category`) and so forth. Sportpage also allows any subtopic to introduce new variables unknown in sibling or parent topic templates.

Preloaded and Cached Objects

The `BasicPageHandler` adds several dot-variables to the context before the configuration variables are parsed. These include a `VelocityCalendar` set to the current time, an application-scoped keyword/value map, and utility objects for fetching database and xml items, and for creating arbitrary java objects.

The `BasicPageHandler` creates a `$date` variable that defaults to Mountain Standard Time ... this can be set to any valid timezone code using the `timezone` property ... and yes, if you want to get perverse, you can set different time zones for different topic areas (ie `sports.bobsleigh.history.timezone=PST`)

SportPage also supports preloading of Javabeans through the configuration file. When dot-variables end in `.bean`, SportPage will interpret the value as the `String` value of the fully qualified class name and create an instance of that class under the variable name. For example, `default.now.bean` set to `java.util.Date` will provide a global variable `$now` containing a Java date object; `sports.venue.now.bean` would only define this value in the `/sports/venue` subtopic. (See Section , “Adding Javabeans”)

Caching Objects

Suppose you have a large collection of XML objects that may or may not be read into sections of pages throughout the site. It would be highly inefficient to have these objects all preloaded, and too awkward and clunky to load and parse each of them on every request; the general pattern of use for such items are that a small set will be used frequently, and the most recently used are the most likely to be requested by subsequent visitors to the site. This latter rule is especially evident with news sites where the resource describes some topical event.

The SportPage solution is a “most-recently-used (MRU) time-to-live (TTL) cache map”, a global lookup table matching item requests to the objects requested, but where inactivity will cause an object to be removed from the cache after some pre-set time lapse. The refresh is such that the next edition is loaded and then stored, so previous objects that are still in scope are still valid for template display. It’s kind of like the afterimages you see watching moving objects while on acid: The data item you get may be a few seconds stale, even while the new one is loading.

For example, given the URI `/fencing/legends/damocles` which expects to be rendered through `fencing.legends.page` set to the template `/fencing/legends.vm`, the template begins with the line

```
$bio = $xml.dom($path)
```

`$bio` now contains the JDOM object.

The `$xml` object uses the expiring TTL cache map, and in the above case, it sees a call to `dom( "/fencing/legends/damocles.xml" )` which it finds in `$dom.home/fencing/legends/damocles.xml` and retains until the association times out. This allows the XML file to be re-parsed for the subsequent calls while handling peak bursts of site activity.

SportPage also provides a renewing MRU cache map for retaining static or self-renewing items such as database result sets. These objects stay in the map so long as they are used and start to expire only after the most recent access.

Adding Javabeans

Sometimes you just need to add some bit of Java to the page and you don’t want, or can’t involve java developers. You know the classname, the class is available to your application (through system libraries or being added to the `WEB-INF` directories) and all you want to do is reference it. This is the function of the `$javabeans` objects.

The BeanFactory `$javabeans` object provides a simplified equivalent to the `<jsp:useBean>` and `<jsp:setProperty>` features in JSP, allowing page designers to add arbitrary Java objects to their templates without requiring updates to the Java webapp libraries or any other java programmer intervention.

To support Java beans, SportPage adds two context utility objects, `$params` and `$javabeans`:

`$params`

this is a class borrowed from Turbine. This is a java Map object populated with GET/POST or PATH_INFO parameters. Access methods provide ways to get at java primitive types (getBoolean, getBigInteger...)

This utility object is often used in conjunction with JavaBeans to set bean properties with one call just as you would normally do in JSP:

```
$params.setProperties( $mybean )
```

`$javabeans`

this is a `BeanFactory` which addresses performance issues with trying to instance new objects on every Velocity template call; unlike JSP, the BeanFactory leaves the scope of the object to the bean itself.

The bean is created according to the following method search sequence:

1. `getInstance(ExtendedProperties)` method which has access to the server configuration.
2. `getInstance(Context)` method that can be used to detect the `HttpSession`, the `$path` or whatever other clue is useful in the page context to decide how to and when to create a new bean.
3. If the context method fails, it will then try `getInstance()`. This allows beans to be implemented as singletons (ie only one bean per JVM)
4. if the singleton method fails, it will then try for a null constructor, the standard constructor for java beans. In this way, any class that can be included in JSP using the `usebean` tag can be included under a `request` scope in a Velocity page; if the need arises (and it probably will) this will be expanded to allow constructors to instance a bean with page and/or webapp scope.[3]

Inside a Velocity template, we can use constructs like

```
#set ($now = $javabean.getBean("java.util.Date"))
```

or, if the bean supports it, have a form handler selecting weather cities and use

```
#set ($weather = $javabean.getBean($context, "cbc.utilities.weather" ))
$params.setProperties( $weather )
```

Pre-Loading Java Beans

Any configuration file dot-variables ending in “.bean” will be interpreted as a fully-qualified Java classname and instanced as a Java bean; like any dot-property, you can specify these per-topic and inherit bean variables from a parent topic

[3]Java beans can also be created under an `applications` scope by attaching bean instances to the `$global` application configuration `Map` object. Because Velocity templates are all, to Java, the same servlet, there is no equivalent to a `page` scope without extra programming to cache the object based on the `$path`; examples of such caching can be found in the `JDOMFile` source code.

```
default.weather.bean=cbc.utilities.weather
```

will define a global `$weather` variable using the specified class

```
sports.luge.history.weather.bean=cbc.utilities.weather_history
```

would mean that, in sports/luge/history *only*, the $weather refers to a different class of object.

Caution

Because dot-variable and `BeanFactor` constructors are evaluated as each template is accessed, they are expensive to invoke; the beans should cache themselves or be singleton objects where ever practical, and constructors should be lightening quick.

Special Beans

SportPage supports two special property bean types, XML and SQL beans, distinguished by properties ending in `.xml` and `.sql`. These properties are trapped by the `BasicPageHandler` and the associated strings used to create `DOMFile` and `DBCacheBean` objects.

Defining PageHandlers

The first rule on defining new page handlers is “don’t”. In most cases, page handlers we created were later deprecated and replaced with the more flexible topic/context manipulations. There are only two reasons to define a new page handler: Special path handling and context manipulations.

Suppose you need special default handling for the `$path` (the `PATHINFO` after the servlet URI) that must occur before any rendering of the page takes place; SportPage includes automatic handling for XML paths, but if you wanted to do something funky with `.swf` extensions (dynamically generated Flash animations based on real-time data?), this would be a good case for a new page handler.

`BasicPageHandler` is a good example of the first case: We needed to test the URI path to see if it is a directory, and if so, swap the `$index` page over the `$page` variable.

A good bad-example is AthletesPageHandler, originally intended to interpret the path as an XML file, returning the JDOM model for the template to pick apart. We soon realized we’d want general purpose XML everywhere, so there was no need for a special topic (directory path) just for XML. PageHandlers attach the pre-rendering to a *path*, not to a purpose.

The second reason for page handlers is to be able to fiddle with the context, for example to dynamically change layout pages or other items.

A quasi-bad example of the second reason for PageHandlers is the `LineupPageHandler`[4] which was originally used to provide access to the Newsworld presentation template classes used by the CBC to retrieve headlines and stories from their news system. Why a bad idea? The same thing could be done using a property bean and, if they ever want to dump it, they won’t need a java programmer.

[4] In science, nothing is ever a complete failure. It can always be used as a bad example.

Templates and Database Access

Dot-variables ending in `.sql` create `DBCacheBean` objects in the MRU cache. In English, this means the results of the SQL queries specified by these dot-variables will be periodically re-fetched so long as the web traffic asks for it; when the web traffic stops, SportPage will stop refreshing and remove the cached result set, rebuilding it the next time the query is requested.

To access any data table, you must first define SQL statements describing each unique table; this can be done using SQL “views” to distill complicated queries down to simple `SELECT` statements. These statements are bound to special dot-variables in the `sportpage.conf` which also means that you can have the same variable name being bound to different select statements in different topic areas.

The general form of the dot-variable definition is

```
topic.varname.sql=SELECT cols FROM table WHERE constraints
```

SportPage will detect the `.sql` and create a `DBCacheBean` for this request; the bean will periodically refresh the query every `dbcache.period` seconds, expiring the query completely if it is not accessed again within a preset time (set with `mru.timeout`).

For example

```
sports.hockey.standings.sql=SELECT * FROM hockey_standings
```

will define the variable `$standings` in any template within the topic `/sports/hockey`.

The `$standings` object has two access methods: `$standings.columns` and `$standings.results`; the first is a list of column names, the second a list of value lists.

Example 2. Rendering `DBCacheBean`

A generic template to render *all* SQL result objects would go something like this:

```
<table>
<tr>
   #foreach ($col in $standings.columns)
   <th>$col</th>
   #end
</tr>
#foreach ($row in $standings.results)
<tr>
   #foreach ($val in $row)
   <td>$val</td>
   #end
</tr>
#end
</table>
```

If you need to re-arrange or rename columns, it’s best to do this in the SQL (so the database does all the work) but you can do a bit of processing on the columns or the results list; these are returned as Java

`ArrayLists`, which provides methods to access specific items by number, so you could hunt for the column number of a particular column and then sift the results for that column number. It's much easier and less processing to simply define a new SQL object, but the downside is that the servlet container must be restarted to load any changes in the dot-variable definitions.

Template XML Handling

Arbitrary XML files are loaded by the `$xml.dom(filename)` method, an instance of the `DOMFileFactory` class. The class of document object is determined by the `dom.data.class` property and defaults to a SportPage `JDOMFile` object.

`JDOMFile` provides methods to query, transform and output the XML from within a template (see Sportwire JDOMFile API Docs[14]) and uses the TTL cache to temporarily store parsed documents.

Querying XML Documents

XML is queried by retrieving the JDOM `Document` from the `JDOMFile` object and applying the standard JDOM access methods. XPath methods are on the TODO list.

Transforming XML

To transform an XML file, the file must be first loaded by the `$xml` object. For example, if you're using the pathinfo to specify the XML file, you would insert the XML file specified by the path relative to dom.home with the following template code:

```
#set ($dom = $xml.dom( $path ))
```

The `transform(xslfile)` method creates a new `JDOMFile` object:

```
#set ($d = $dom.transform( "/path/to/transform.xsl" ))
```

You can then insert the result via the XML method to include the XML text inline:

```
$d.xml
```

If you want to save space, this can be collapsed into

```
$xml.dom( $path ).transform( "/path/to/transform.xsl" ).xml
```

Note

Since `transform` returns the newly transformed document, you can chain calls, appling each XSL file to the result of the previous transform.

[14] http://sportwire.sourceforge.net/api/ca/cbc/sportwire/servlet/data/JDOMFile.html

As a side benefit, if you want to look at any XML file as a nicely formatted block of text, you create a webpage which simply says ...

```
   $xml.dom( $path ).xml
```

Example Applications

This section describes some special effect recipes developed while working on the Olympics website.

The Simplest Templates

The SportPage `webapp` directory contains some very skeletal example templates that demonstrate some of the features of the system.

Example 3. Sample Layout Template

At the very basic level, a website consists of a `default.layout` and `default.page` templates where the layout sets up the design of the browser window and includes the page at a suitable point:

```
<html>
<head><title>Hello World</title></head>
<body>
#parse($page)
</body>
</html>
```

You could, of course, define all sorts of contant strings in the `WEB-INF/sportpage.conf` to set filenames, banner images or whatever you like to fill out that default layout. The only essential detail is the call to parse the `$page`.

Ok, that sets up the stunning graphic design of our site, now we need some content:

Example 4. Sample Page Template

The simplest page would be static, but to make it just a little interesting, we want to take the path information and use that to fetch another file to insert. The caveat here is that the URI is human-oriented; it has no extension, and there are several extensions listed in the defaults. The page template needs to handle the implied extension, and also must handle a missing page instance. To do this, SportPage includes a library macro method `#insert( path )` which will test that the path exists and insert the error page if it does not:

```
## handle missing page inserts
#macro (insert $path)
#set ($p = false)
#set ($p = $path.ext)
#if ($p)
#parse($p)
#else
```

```
#parse($missingpage)
#end
#end
```

This macro simplifies the page template to become:

```
<table>
<tr><th>Hello Page</th></tr>
<tr><td>
#insert($path)
</td></tr>
</table>
```

In the macro, the `PathBean.ext` method will identify any file matching either the path as given, or some variation across all the template paths and extensions.

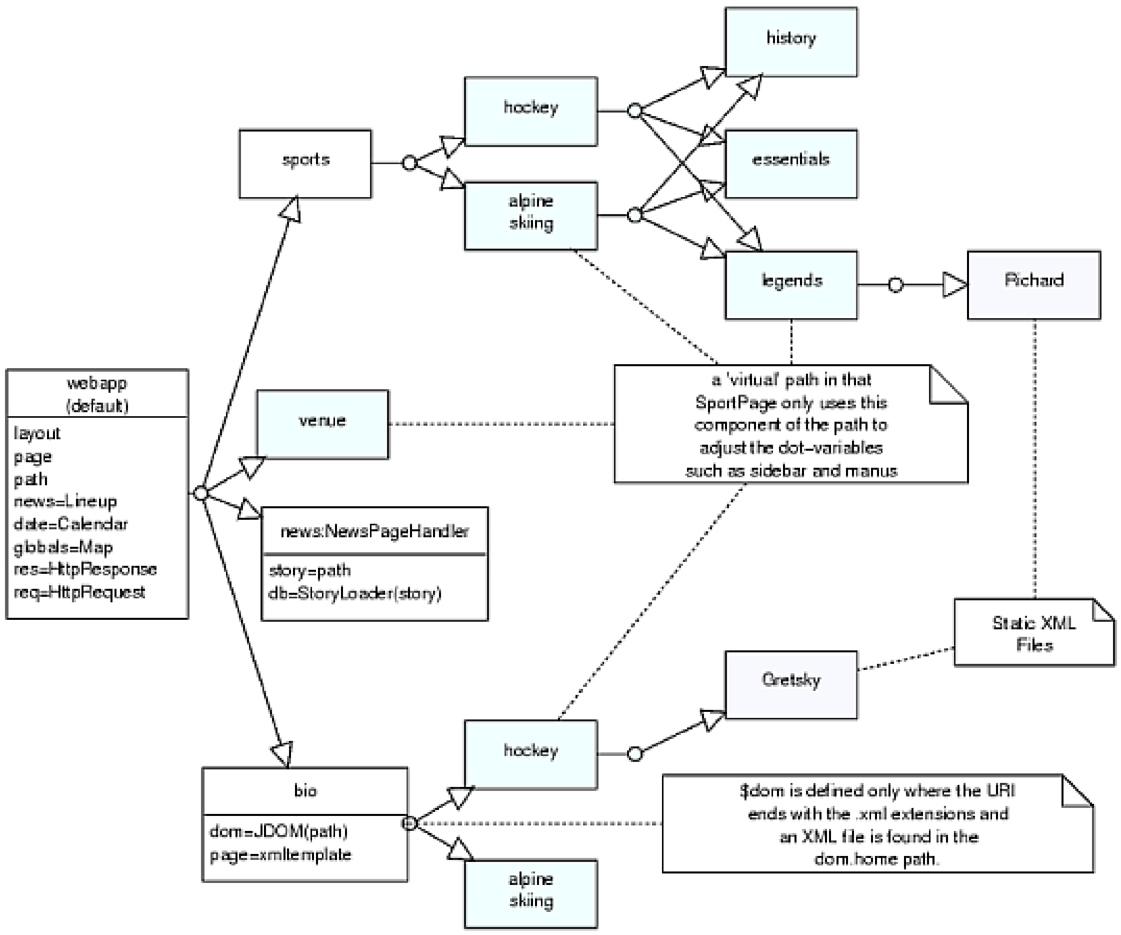
That's it, our first SportPage website!

A Complex Example: Topic-based Website Structure

The CBC 2002 Winter Olympics[15] is arranged in a topic hierarchy as shown in Figure 1. In this site, the default page handler provides only a switch between index and document page templates. Exceptions to this are the `/olympics/athletes` area and the `/news` topic. Athlete URI paths actually specify an XML document (the athlete bio) and the template page is replaced (using a dot-variable) with a form page that fetches and reports the path as a JDOM object. The news path is used as the target of the headline links and uses the dot-variable to specifie a news item "fullstory" template where the specific story key is identified by the path; the URI `/olympics/news/kelly001` pulls the story `kelly001` from the Newsworld database.

Figure 1.

[15] http://cbc.ca/olympics



Adding External Java Objects

Adding CBC Newsworld news items provides an example of implementing arbitrary java objects. In this situation, the Newsworld classes provide methods to access the key strings for news headline items.

Example 5. Adding Newsworld's `Lineup`

The following describes creating new story pages using the `$path` to get at the story key, and also formats main page headlines to make links relative to our own webapp `/olympics/news`.

The first necessary step is to add the Lineup object to the system through a dot-variable. Because we need to cache lineups on a per-category basis, we will want to define a category string (and possibly others within other topic contexts) and we need to create a wrapper class that will be able to store these cached items in the system `$global` properties; this is the function of the SportWire `TemplateLineup` class. The categories and this bean class are added to the `sportpage.conf`:

```
##############################################
# Install the CBC presentation.Template lineups
default.lineup.category=olympics
hockey.lineup.category=olympichockey
default.news.bean=ca.cbc.sportwire.servlet.data.TemplateLineup
```

We are free to make our own URIs because Brian provided an alternate `getSlug()` method that returns only the story key string slug:

1. insert the following into the home page template to render the headlines:

   ```
   #foreach( $story in $news.lineup.getStories(1,3) )

    <span class="HEADLINE">$story.headline</font>
     #if( $story.hasTopStoryImage() )
       <img src="$story.topStoryImage"
            height="100" width="100" align="right">
     #end
     <p class="LEADERS">
     <span class="ABSTRACT">$story.abstract</span>
     <span class="ABSTRACTLINK">
        <a href="/olympics/news/$story.slug">
        <nobr>full&nbsp;story</nobr></a>
     </span>
     </p>
     <br clear="all">

   #end
     <table class="HEADLINES" width="100%">
       <tr>
         <td class="HEADLINES" colspan="2">
         <span class="HEADLINEBANNER">H E A D L I N E S</span>
         </td>
       <tr>
         <td class="HEADLINES">
           <ul>
              #foreach( $story in $news.lineup.getStories(4,10) )
              <li><a href="/olympics/news/$story.slug">
                  $story.headline</a></li>
              #end
            </ul>
          </td>
       </tr>
     </table>
   ```

   This much makes the lineup appear according to our own rules and are linked to our own full-story URI.

2. define the fullstory template in `WEB-INF/sportpage.conf` and add the `StoryLoader` object such that it's only available in the `/news/` topic context:

   ```
   news.page=/default/fullstory.vm
   news.db.bean=cbc.template.story.StoryLoader
   ```

3. The template `fullstory.vm` must fetch the `Story` object; this is done using `StoryLoader` and trimming the `/news/` from the `$path`. Keep in mind this example alters the `$path` object, so you can't use it further down the page. `$base` should get set to `news`.

   ```
   #set ($base=$path.remove(0))      ## trim path to /olympics/story99
   #set ($slug = $path)
   #if ($slug == "")                ## prevent accidental visits
      $res.sendRedirect("/olympics")
   #end
   #set ($story = $db.load($slug))
   ```

   then follow with your own rendering of the `$story` object:

   ```
   <span class="HEADER">$story.headline</span>
   <br>
   <span class="SUBHEADER">Last Updated: $story.lastUpdated</span>
   <br>
   <span class="SUBHEADER">$story.subheadline</span>
   <p>$story.dateline - $story.getParagraph(1)</p>
   ...
   ```

Translating from JSP `usebean`

Incorporating the Newsworld weather forecast Java bean provides an example of translating JSP style to Sportpage. The original JSP code would read something like this:

```
<jsp:useBean id="weather" scope="page"
class="cbc.beans.ShortWeatherBean"/>
<jsp:setProperty name="weather" property="query" value="Toronto,ON"/>
<%=weather.getWeather()%>
```

The SportPage equivalent would use the `$javabeans` utility to create the weather object, then use access methods to fetch the weather forecast:

```
#set ($forecast = $javabeans.getBean("cbc.beans.ShortWeatherBean"));
$forecast.setQuery("Salt Lake,UT")
...
<p>Weather for Salt Lake City: $forecast.weather</p>
```

Of course, if the bean doesn't cache it's results, it's going to run like arctic molasses.

Reports and Troubleshooting

Although servlets add a logging system and Velocity adds another, I don't like either, so SportPage includes the Log4J logging system and is peppered throughout with debug, warning, info and critical error messages that will be written out according to the specifications in the `WEB-INF/log4j.conf` file.

By default, velocity errors, sportpage errors and other application messages are written to `WEB-INF/logs` and the first action in any template system debugging should be to follow those logs while the request is being issued or while the webapp is loading. You can set this log file to another location by editing the `log4j.appender.A1.File` setting.

Web Application Log

The web application log file records all messages trapped by the SportWire software. This is the file generated by the `Log4J` system and can be tuned to increase the level of reporting for specific SportPage components.

Velocity Log

The velocity log file records all messages coming out of Velocity itself. The location of this file is specified by the `velocity.log.filename` property in the `WEB-INF/sportpage.conf` system.

Servlet Log

Your local system administrators should be recording the output from the servlet container. This file will record critical messages such as stack traces, items which neither velocity or the application are able to trap.

Template Variable Reference

SportPage expects and provides a small set of template variables. As much as possible, SportPage does not dictate how these variables are used or what value types they may have; technology should serve the application designers, not enslave them.

Required Variables

There is a very small set of variables which must be defined in the `WEB-INF/sportpage.conf` configuration file. These are primarily concerned with error recovery and index-page handling and were thus hard-coded into the SportPage design. These required values are summarized in table Table 1