Developing Applications with WireHose (Mac OS X)

Table of Contents

Table of Contents

Table of Contents

Table of Contents

About this document

WireHose Server gives developers the power to create fast, personalizable web applications that categorize, index and deliver any media type, like video clips, stock quotes or legacy data. It features a simple, flexible API for adding new personalizable components and resource types, making it perfect for creating enterprise portals and high−traffic internet sites.

Why read this document

This document teaches you how to build web applications using WireHose by leading you through the process of developing a personalizable news aggregator. It covers the overall architecture of a WireHose application, key portions of the WireHose API, and discusses the use of WireHose tools. It also includes information about advanced WireHose topics.

It is available in two editions, one for developers using Mac OS X, and one for developers using Windows.

What you should already know

You should be familiar with building and running applications with WebObjects.

Contents

Crawling feeds

Describes the process of building a crawler which will import items from feeds on a regular basis.

WireHose layout concepts

Describes key WireHose layout components such as pages, wrappers, areas and the layout dictionary.

Customizing the user interface

Utilizes WireHose layout features to control the application's appearance and behavior with minimal code.

Adding personalization

Works through allowing users to sign up for a new account and personalizing their page.

Finishing the user interface

Adding navigation and other niceties to the sample application.

Further exploration

Summarizes several WireHose features which were not covered in this document.

Related documentation

To get an overall feel for the various parts of the WireHose API, see the Developer Overview, which provides brief descriptions of important WireHose concepts and classes.

The Java API Reference is the authoritative source for information about WireHose concepts and classes.

See the Properties Reference for details about controlling the runtime behavior of WireHose applications through command−line parameters.

Database Setup describes how to set up WireHose to work with nearly any database.

For details about regular expressions support in WireHose, see the Quick Start and Pattern Reference.

About WireHose

WireHose gives Java developers the power to create fast, personalizable web applications that categorize, index and deliver any media type, like video clips, stock quotes or legacy data. It features a simple, flexible API for adding new personalizable components and resource types, making it perfect for creating enterprise portals in addition to high−traffic internet sites.

WireHose features

Content management

WireHose provides a powerful, flexible foundation for building content management systems. Managed content as well as legacy data sources can be categorized and searched by metadata and keyword indexing. Users can easily browse, search and personalize access to any WireHose resources, regardless of data source or content type. Our advanced design enables creation of intelligent metadata with behaviors beyond simple organization into categories.

Personalization

WireHose provides everything you need to create high−performance, scalable personalized applications. Developers can build easy, intuitive, interfaces for users to customize the content and resources of a site within roles−based limits set by the site administrator. Visitors access a personal site that's fully customized to their needs, on any kind of output device, including email, cell phones, pagers or web services.

Access control

Developers can easily integrate a WireHose application with existing authentication and authorization systems to control how users are authenticated and what they can view, edit or delete. WireHose also provides a flexible and powerful system for enforcing roles−based access control to metadata, channels and content, as well as a very powerful group and permission templating capability, which is ideal for creating multiple groups and categories with preassigned permissions, as when creating multiple departments, workgroups or classrooms within an organization.

Dynamic layouts

WireHose deployments can support multiple branded affiliates, as in an application service provider environment, or to allow the user to personalize the look of their page in addition to personalizing its content. The user interface abstraction layer lets developers support multiple output formats, such as XML, HDML, SMIL, RSS, RDF, etc., without duplicating important business logic.

Rapid development

The WireHose frameworks allow developers to build content management and portal applications faster, through the use of templates for rapidly building new applications and reusable components, and a next−generation content management portal API. WireHose provides a consistent interface for managing metadata regardless of back−end data source, allowing developers to focus on relevant business logic rather than implementation details.

Database independence

Databases can be switched during development or deployment without rewriting code. Any database row can become an object which can be categorized and fetched by tags and keywords. Any foreign JDBC, LDAP, JNDI source can be custom queried, and take advantage of finely tunable caching to ensure common requests are not fetched more often than necessary.

WireHose architecture

WireHose is distributed as a collection of compiled frameworks, documentation and sample code. A WireHose application links to the WireHose frameworks, and adds business logic, web component definitions, application and session−level logic, and configuration files. WireHose applications can also be command−line tools or provide web services via SOAP or XML−RPC.

Core frameworks

These are the basic WireHose frameworks:

WireHoseBase
: Contains basic enterprise object interfaces and classes used by all WireHose projects, as well as utility classes used internally by WireHose.

WireHoseLayoutSupport
: Contains core web application layout classes, including application and session−level logic. Also provides reusable components, page−level components and direct action interfaces for WireHose pages.

WireHoseEngageSupport
: An optional framework which provides roles−based access control to taggable objects and a templating system for creating hierarchical collections of tags.

Additional frameworks

WireHose also provides some optional frameworks:

WHOpenBasePrototypes
: Contains enterprise object attribute prototype definitions for OpenBase.

WHFrontBasePrototypes
: Contains enterprise object attribute prototype definitions for FrontBase.

WireHoseWOBuilderBindings
: A fake framework and associated Project Builder project which helps WebObjects Builder properly display the custom bindings available in WireHose components.

Application structure

While you can build simple WireHose applications in a single project −− and that is the approach taken in this tutorial −− to achieve maximum code reuse, you'll probably want to structure your application as a collection of frameworks. This approach allows you to build multiple applications which share common code without duplication.

Here's an example of an extremely factored approach:

User frameworks
: Contain custom user classes and any objects which manage collections of channels for users.

Tag frameworks
: Contain custom tag classes. For example, you may have one framework for workflow tags, and another for tags which model versioning or special access control requirements. Since WireHose provides a unified API for dealing with all types of tags, each collection of tag types can be considered a single extension to an application. These frameworks may include default user interface component definitions, which could be overridden in other frameworks or the application itself.

Channel frameworks
: Contain custom channel types. For example, opinion polls, Amazon searches, legacy calendar access, etc. These frameworks may include default user interface component definitions, which could be overridden in other frameworks or the application itself.

Resource frameworks

Contain custom taggable and indexable objects. For example, bookmarks, pictures, email messages, contacts, movies, news stories, Word documents, spreadsheets −− or any other type of enterprise object. These frameworks may include default user interface component definitions, which could be overridden in other frameworks or the application itself.

Layout frameworks

Contain web components and images which make up various layouts, or skins, for a web application. For example, in addition to having "plain" or "fancy" layouts, you can also define layouts for various XML formats, or even PDF or Flash.

Sample applications

There are a lot of features in WireHose that the Hello World application won't cover. WireHose comes with several other example applications which demonstrate various features.

NewsDemo

NewsDemo is a personalizable news aggregator which combines articles from multiple categories into topics. It provides users the ability to personalize access to news and traffic cams, as well as letting them choose from several possible layouts. NewsDemo was designed to handle the high−traffic personalization needs of television station and newspaper news portals.

NewsDemo also includes an administration interface which allows editors to create collections of pre−built topics for users to choose from.

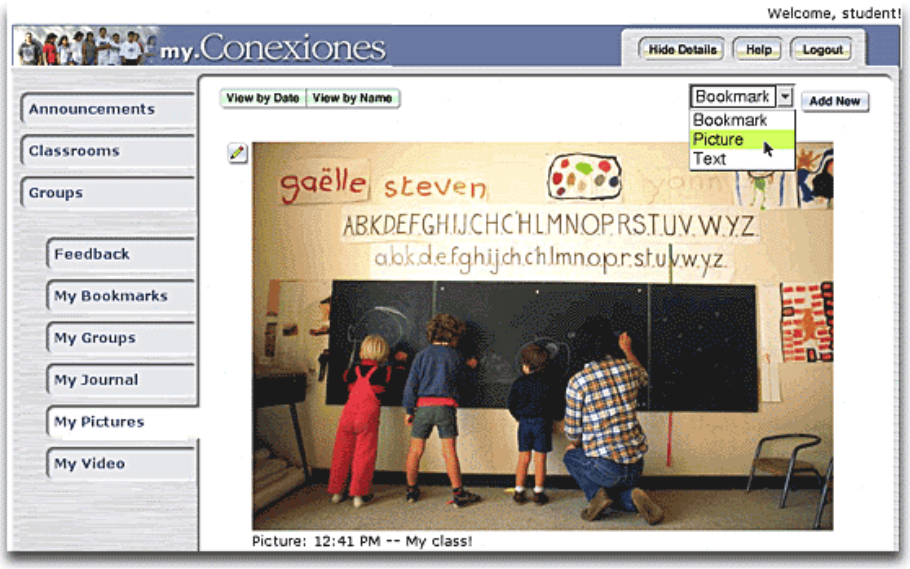
Conexiones

Conexiones gives educators the power to open a world of interactive communication, exchanging resources and ideas between students, parents and teachers.

Students learn better when they're engaged with interactive, compelling content that is easy to navigate, and personalized to their learning level and special interests.

Teachers post assignments and classroom resources. Students browse resources, post homework and participate in classroom research. Parents monitor the student/teacher relationship and participate directly in the learning process. Group participation enriches the learning experience for all involved.

Conexiones involves students in an interactive learning community, where they communicate with other students, teachers and parents to help create their own curriculum, based on their specific academic interests, strengths and weaknesses.



Manage any type of content

Educators, parents and students can enhance learning through interactive displays of animation, video and audio. Word processing, spreadsheets and presentations are also easily incorporated.

Full indexing and categorization

Users can easily track down relevant content in any format by keywords and categories, rapidly connecting them with the resources they're looking for.

Personalization

Users can upload their own content into personal categories such as "My Bookmarks" or "My Pictures", and easily share it with other classroom members or the general public. Everyone has quick access to their own content, classrooms and groups.

Parental access

Parents can easily see how their students are doing, and exchange private feedback with teachers, involving them directly in fulfilling their specific educational needs.

Access control

Access to classrooms, groups and content can be controlled as required, respecting the privacy and special needs of students and staff. Administrators can easily create classrooms with partitioned private and public content, categories and groups.

Flexible classroom and group templates

Teachers can organize the content for their classrooms the way they see fit, and administrators can provide useful defaults. Administrators can define default categories, groups and permissions when creating new classrooms, groups and users. Templates can define any number of sub−categories and groups, and can be easily fine−tuned for individual needs.

Easy to use

Conexiones features an uncluttered interface, simple enough for children and parents to use, yet unleashes the full power of a real content management portal for teachers and administrators.

About Hello World

This document will teach you the fundamentals of building WireHose applications by walking you step by step through the process of putting a sample application together. As is traditional, this application will be called "Hello World".

However, since the WireHose frameworks provide significant functionality out of the box, this Hello World application will do much more than simply echo some text to the screen.

Hello World will provide personalized access to thousands of syndicated headlines, photos, musings and other content, from hundreds of websites across the internet. In other words, you'll build a world−class personalizable aggregator of syndicated content.

Syndicated content

Thousands of websites now distribute their content through more than just a web browser. These sites make summaries of their new content available in XML files. The most common format for these files is known as "RSS", introduced by Netscape and popularized by Userland Software. The RSS file for a website is referred to as a "feed".

There is no consensus on what RSS stands for −− some call it "Rich Site Summary", while others call it "Really Simple Syndication". No matter how you refer to it, though, it is a powerful, yet simple, way to gather and distribute information.

There are several types of RSS files; for this tutorial, we'll be focusing on the simplest format, known as RSS 2.0. A specification for the RSS 2.0 format can be found here.

Aggregators

An aggregator is an application which collects content from diverse sources and presents it in an organized fashion. There are two common types of aggregators, those which run on a desktop and provide content to a single user, and those which run on a server and provide content to multiple users.

Hello World will be a server−based application which provides personalized access to the content distributed in RSS feeds to multiple users.

Browsing and searching

The most common user interface for presenting aggregated content is to let users browse through the individual feeds, perhaps arranged into categories. Hello World will use this interface, with the added ability to browse through the individual items contained within the feeds.

A good aggregator should also allow the user to search the available feeds and items by keywords; since this capability is built into WireHose, Hello World will also provide this function.

Personalization

Since an aggregator may collect items from hundreds or thousands of feeds, it's important to let users select the subset of the available content they're interested in. Otherwise, attempting to keep up with all that information would be like drinking from a firehose!

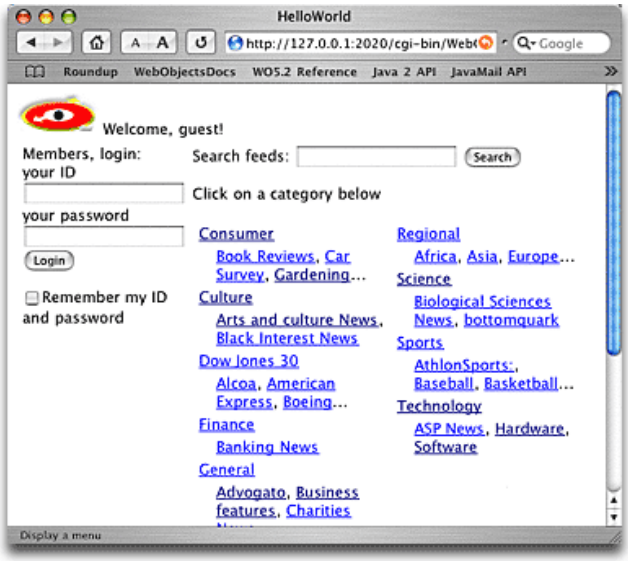
The way personalization is handled in most aggregators is to let users select a collection of feeds they're interested in. The items in these feeds are then presented to the user as a collection of boxes on a page, one per feed, each with a list of the items currently in the feed.

Hello World will go one better than that. Since users will be able to browse through the individual items in the feeds, they'll also be able to personalize access to collections of items, no matter which feed they originated from. In addition, users will be able to easily turn the results of any keyword search into a personalized topic on their page.

What WireHose provides

WireHose provides much of Hello World's functionality out of the box. Content categorization, browsing, searching and personalization are all handled by the WireHose frameworks.

You'll model the database attributes of feeds and items, and provide users the ability to sign up for a new account. You'll also write the feed crawler, which will fetch the individual feeds and import new items into the database. And, of course, you'll have full control over the application's look and feel.



WireHose business logic concepts

Nearly all WebObjects applications revolve around fetching, inserting and modifying data in a database. The enterprise objects frameworks within WebObjects make this possible in a database−independent fashion, allowing you to focus on the business logic within your application rather than the details of a relational database.

Often you will want to be able to organize enterprise objects into categories which users can browse, or search via keywords. This is common in many kinds of applications, including content management, portal, workflow, knowledge management and publishing applications. WireHose was developed specifically to solve these types of problems.

However, once you get a good feeling for what WireHose can do for you, you'll recognize many other scenarios where WireHose can save development time and help create a richer user experience. WireHose provides a powerful toolset which you can use to unify widely varying data such as customer information, email messages or catalog items, in a consistent interface.

Resources

WireHose extends WebObjects so you can create enterprise objects which can be tagged with categories and searched by keywords. Objects are defined as "taggable" and/or "indexable". By convention, objects which are both taggable and indexable are referred to as "resources".

The taggable and indexable capabilities are defined as Java interfaces, called WHTaggable and WHIndexable, respectively. WireHose uses this approach so they can be added to any enterprise object class −− even ones you've already modeled and implemented −− by adding a few relationships and implementing these interfaces.

WireHose provides an assortment of objects and methods which fetch and manage objects which implement a particular interface, rather than those of a specific entity, so any code written to work with one type of taggable object will work with all taggable objects.

Tags

Tags are enterprise objects which can be arranged in an arbitrary hierarchy and assigned to taggable objects. A taggable object can have any number of tags. Tags can be arranged into any hierarchy required. Each tag has a parent tag, and most tags have child tags. (A top tag is its own parent.) WireHose provides methods for retrieving a tag's ancestors, relative ancestors, descendents, relative descendents and leaf descendents.

A tag can be uniquely identified by its tagpath, which is a slash−delimited string indicating its position in the hierarchy. For example, the tagpath "Cats/Black Cats/Budu" identifies the tag named "Budu". You use static methods in the WHTag class create and retrieve tags, and assign them to taggable objects.

Note: You can subclass WHTag to implement access control, workflow or other special applications. For example, the WireHoseEngageSupport framework defines several WHTag subclasses which implement roles−based access control for taggable objects −− including WHEngageTag, which defines tags that are taggable.

Personalization

Users

WHUser is the parent entity for all WireHose users. The login "guest" is reserved for the guest user, which is who users are until they login. Guest users are generally created and retrieved automatically; you can override the default guest user creation behavior through a delegate method.

Channels

Channels are defined by the WHChannel interface. Each user may have one or more channels, which represent objects which have been personalized, such as fetchers, polls, stock trackers, etc. WireHose also defines global channels, which belong to all users. WHUser provides several methods for filtering global and user channels.

Channels belong to individual users, and each channel keeps a reference to the factory which created it. Channels are bound to a specific area on the user's page through the `areaName` property; the default area name is "main."

The order in which channels appear within an area is determined by the channel's `importance` property; channels with lower importance appear before higher importance.

Subclasses of WHUser can override the allUserChannels method to include channels which are not modeled as part of the base "channels" relationship.

Channel factories

Channel factories, defined by the WHChannelFactory interface, are objects which create channels for users. Typically these are created by an administrator and contain appropriate preset values, and are presented to users in a checkbox list. They are also often used to cache expensive calculations or database fetches. They can also be used to provide default settings for channels that can be overridden on a per−user basis.

Fetchers

Fetchers fetch resources from a remote source, such as a database or web service. Fetches are considered expensive operations, so fetchers cache the results of their fetches.

WireHose provides several classes which work together to fetch and cache objects, and uses notifications to ensure that caches are invalidated when necessary. Often when working with fetchers, you will be fetching objects which support a particular interface such as WHTaggable, so fetchers provide methods to filter the returned objects based on entity or interface names.

There are several types of fetchers which act as channels for users. The most commonly used fetchers in WireHose are those that retrieve taggable and indexable objects based on optional and required tags and keywords.

Getting started

The first thing you'll do is create the Hello World project. WireHose includes Project Builder templates which include the default frameworks and files needed by all WireHose applications.

The next step will be to set up the database for Hello World. This document includes specific instructions for setting up OpenBase or FrontBase, but you can use any database supported by WebObjects.

Before you begin

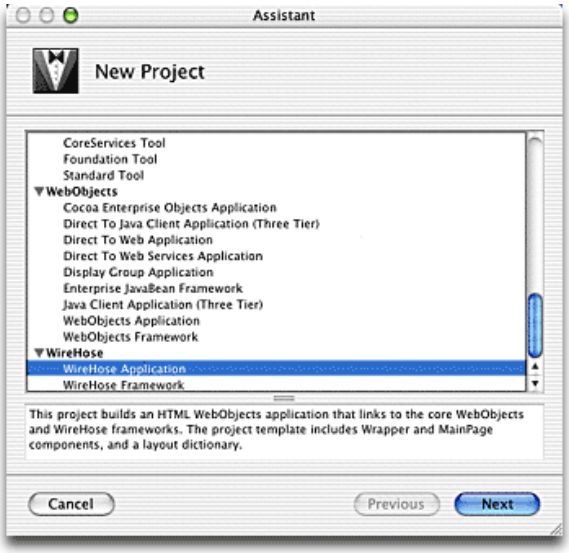
Before creating any WireHose projects, be sure you have done these steps:

1. Install WireHose on your system.
2. Copy the **WireHoseExtras** folder to your hard drive (for example, in your home directory).
3. Copy the contents of **WireHoseExtras/Developer** to **~/Developer**. This folder contains Project Builder templates for creating WireHose applications, frameworks, components, and more.

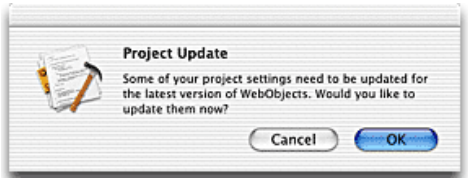
Creating the project

Now you'll create the project. WireHose includes Project Builder templates which include the default frameworks and files needed by all WireHose applications.

1. Launch Project Builder, and choose **New Project...** from the **File** menu.
2. Scroll down to **WireHose Application** template, and click **Next**.



3. Name the new project "HelloWorld", and click **Finish**.
4. If you are asked "Some of your project settings need to be updated for the latest version of WebObjects. Would you like to update them now?", click **OK**.



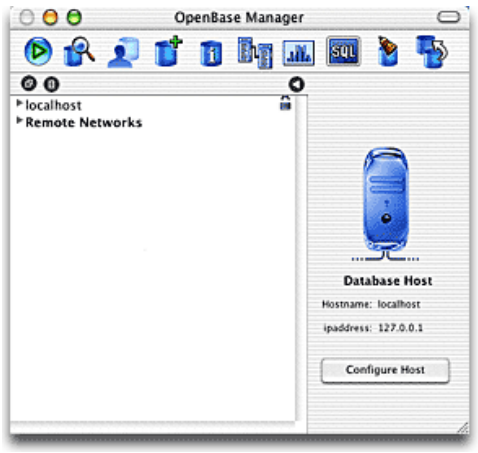
Creating the database

The WireHose frameworks are designed to work with any database supported by WebObjects, and offer several features to make cross−database development and deployment easier.

This document includes specific instructions for setting up OpenBase or FrontBase, but you can use any database supported by WebObjects. See the WireHose documentation more information on using WireHose with other databases.

Setting up OpenBase

1. Launch OpenBaseManager.

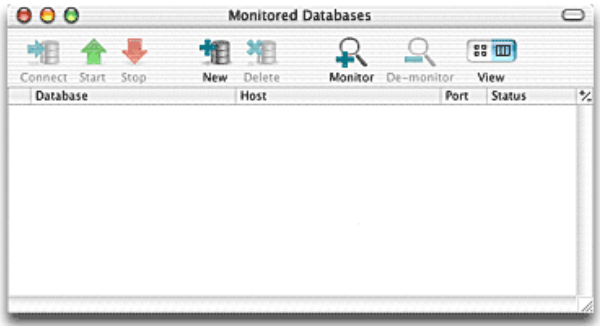


2. Choose **New...** from the **Database** menu.
3. Name the database **HelloWorld**, and click **Set**.

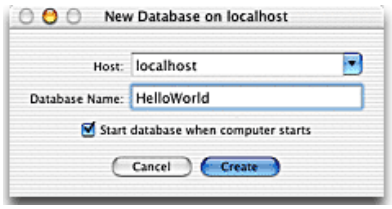
4. Choose **Start** from the **Database** menu.
5. Choose **Run SQL Script** from the **Tools** menu.
6. Select **WireHoseBase.sql** in the **WireHoseExtras/SampleData/OpenBase/** folder, and click **Open**.

Setting up FrontBase

1. Launch FrontBaseManager.

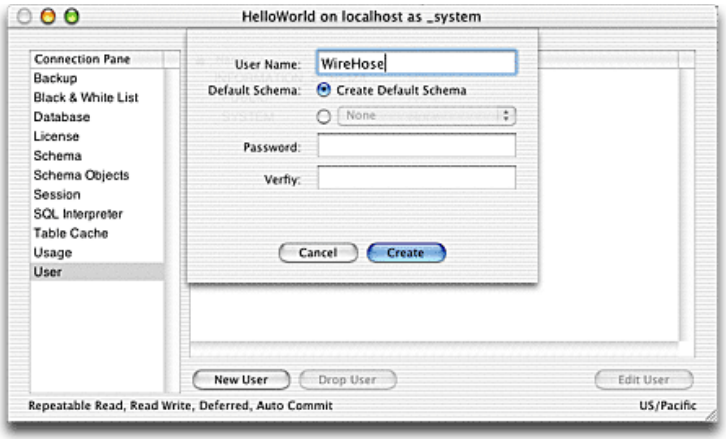


2. Choose **New Database...** from the **File** menu.
3. Create the database on localhost, call it **HelloWorld**, and click **Create**.



4. Select the **HelloWorld** database in the Monitored Databases window, and click **Connect**.
5. Select the user **_system**, and click **Open**.

6. Select **User** in the connection pane, and click **New User**.
7. Set the user name to **HelloWorld** and click **Create**.



8. Close the connection window.
9. Select the **HelloWorld** database in the Monitored Databases window, and click **Connect**.
10. Select the user **HelloWorld**, and click **Open**.

11. Select **SQL Interpreter** in the connection pane, and click **Execute File**.
12. Select **WireHoseBase.sql** in the **WireHoseExtras/SampleData/FrontBase/** folder, and click **Open**.

13. Close the connection window.

Configuring Hello World for the database

The WireHose frameworks are database independent; a WireHose application can use any database which is compatible with WebObjects. There are two primary techniques which are used to achieve this: replacing the adaptor dictionary, and the use of attribute prototypes.

Note: For maximum performance, WireHose fetchers use custom SQL, some of which may be incompatible with some databases. Use of this SQL can be controlled by several system properties. In general, any database which supports subselects should work properly with WireHose, though databases that support SQL set operators such as INTERSECT and EXCEPT (or MINUS) will perform better.

The adaptor dictionary

Each EOModel in a WebObjects application contains information in its index.eomodeld file called the "connection dictionary", which tells WebObjects which database on which host to connect to, which JDBC driver to use, etc.

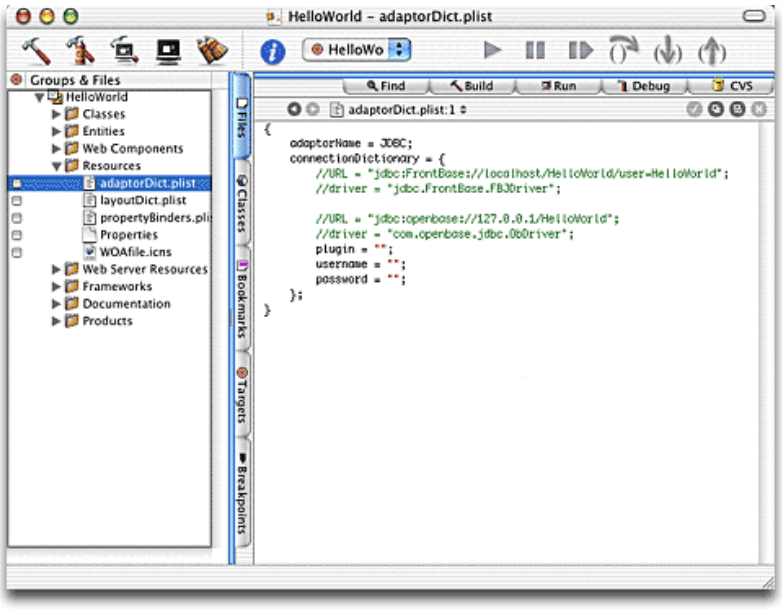
WireHose can replace the database adaptor and connection dictionary on the fly when each EOModel is loaded. This allows you to switch between multiple databases by specifying a different adaptor dictionary on the command line when launching your application, as, for example, during development versus deployment.

An adaptor dictionary file is plain text in plist format, with two keys, "adaptorName" and "connectionDictionary". There are sample adaptor dictionaries for FrontBase, OpenBase and Oracle located in the `WireHoseExtras/SampleData/sql` directory.

Note: You can use the WHAdaptorDict property to specify the name of the adaptor dictionary to use at runtime. You can specify either the name of an adaptor dictionary in your application's Bundle Resources, or the full pathname to the desired file.

To set up the adaptor dictionary for Hello World:

1. Select the **adaptorDict.plist** file in the **Resources** group in the Files pane in the Hello World project.



2. Uncomment the appropriate lines for your database.

If you're using OpenBase:

```
URL = "jdbc:openbase://127.0.0.1/HelloWorld";
driver = "com.openbase.jdbc.ObDriver";
```

If you're using FrontBase:

```
URL = "jdbc:FrontBase://localhost/HelloWorld/user=HelloWorld";
driver = "jdbc.FrontBase.FBJDriver";
```

Database and prototype frameworks

The WireHose Application project template automatically includes references to OpenBase and FrontBase−specific frameworks. You'll need to add the frameworks for your database to the Application Server target so they'll get loaded at runtime.

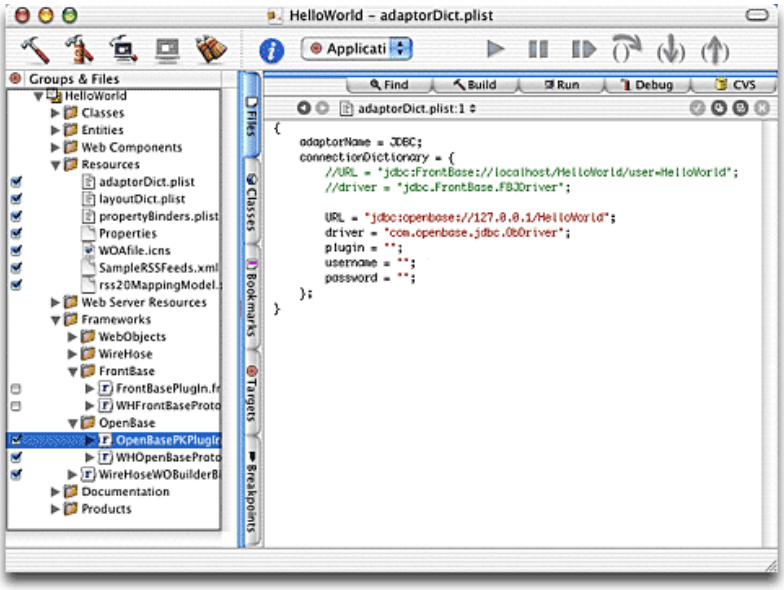
All WireHose−specific entity attributes are defined by a handful of attribute prototypes. Using prototypes allows you to change the definition of every WireHose entity in a single place, no matter which model or framework they reside in (including entities defined in the WireHoseBase framework).

See Apple's Using EOModeler documentation for details about creating prototype definitions. It is a convention in WireHose applications to place the prototypes model in a framework, so that adding this framework to any application will cause WebObjects to use your prototype definitions instead of the ones defined in the WireHoseBase framework.

Note: To use a prototypes framework in a command−line tool, make sure there is entry in your classpath which points to the prototypes framework directory as there won't be a jar file to point to, e.g., "...:/Library/MyPrototypes.framework:..."

To add the database frameworks to Hello World's Application Server target:

1. Make **Application Server** the active target.
2. Expand the **Frameworks** group in the Files pane.
3. Expand the group for your database.
4. Check the boxes next to the frameworks for your database.



If you're using OpenBase:

`OpenBasePKPlugIn.framework`
`WHOpenBasePrototypes.framework`

If you're using FrontBase:

`FrontBasePlugIn.framework`
`WHFrontBasePrototypes.framework`

5. Make **HelloWorld** the active target.

Modeling the data

The key step in developing any application is modeling the data, and WireHose applications are no different. In this section you'll use templates to model RSS feeds and items as enterprise objects which can be categorized by tags and indexed by keywords.

An XML mapping model will be used to extract items from an RSS feed. A list of sample feeds is provided in RSS format, with each item representing a feed.

WireHose provides several utility methods to perform tasks such as fetching XML content from remote URLs, and inserting resources into the database. You'll use these to write a feed crawler, which will fetch feeds, extract items, and insert them into the database.

In its purest form, an RSS feed contains a list of items, each of which has a title, link and optional description. There is often much more information available in the feed, such as the date each item was published, but to keep this application simple, Hello World will only deal with titles, links and descriptions.

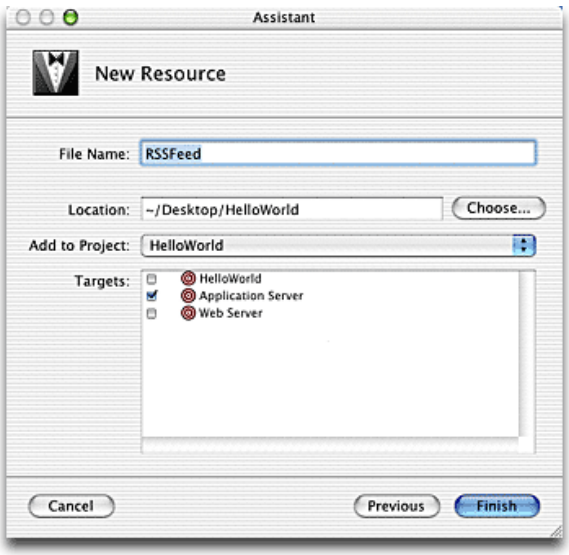
Modeling feeds

WireHose includes Project Builder templates to create new resources from scratch, which is the approach we'll take in this tutorial. See the reference documentation for WHTaggable and WHIndexable for details about adding support to existing enterprise objects.

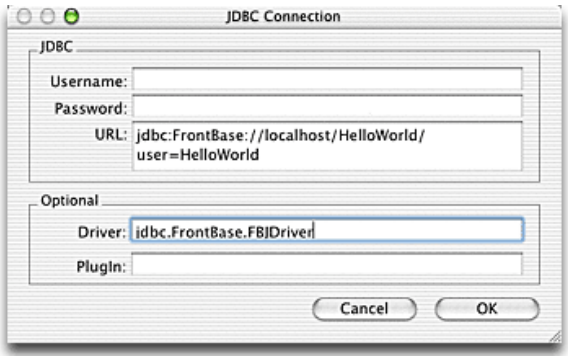
Creating RSSFeed

1. Expand the **Entities** group in the Files pane.
2. Choose **New File...** from the **File** menu. Scroll down to the WireHose **Resource** template and click **Next**.

3. Name it **RSSFeed**, add it to the Application Server target in the Hello World project, and click **Finish**.



4. Open the **RSSFeed.eomodeld** file in EOModeler.
5. Choose **Set Adaptor Info...** from the **Model** menu.
6. Set the URL and driver as entered in your adaptor dictionary:



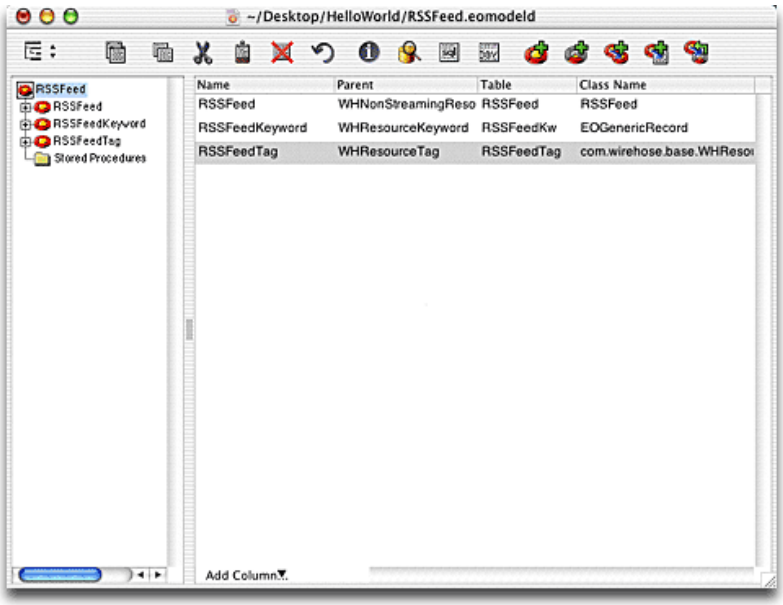
If you're using OpenBase:

Set the URL to **jdbc:openbase://127.0.0.1/HelloWorld** and the driver to **com.openbase.jdbc.ObDriver**

If you're using FrontBase:

Set the URL to **jdbc:FrontBase://localhost/HelloWorld/user=HelloWorld** and the driver to **jdbc.FrontBase.FBJDriver**

7. Click **OK**.
8. Change the name of the **MyResource** entity to **RSSFeed**, its table to **RSSFeed**, and its class name to **RSSFeed**.
9. Change the name of the **MyResourceKeyword** entity to **RSSFeedKeyword**, and its table to **RSSFeedKw**.
10. Change the name of the **MyResourceTag** entity to **RSSFeedTag**, and its table to **RSSFeedTag**.



Adding attributes

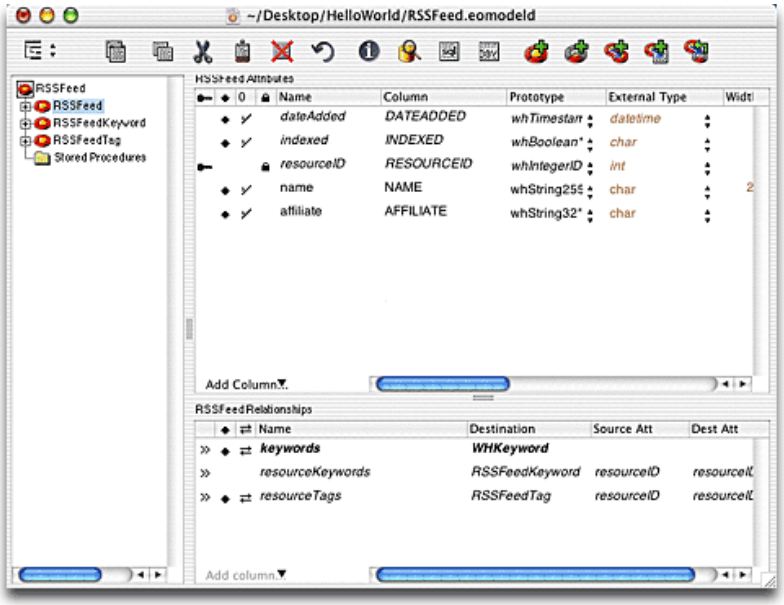
The new resource template includes several attributes which are part of the WHTaggable and WHIndexable interface. The `dateAdded` attribute is common to both interfaces, and allows users to search for recently inserted resources. The `indexed` property and the `resourceKeywords` and `keywords` relationships provide the ability to search by keywords, while `resourceTags` is used to provide a to−many relationship to available tags. The `affiliate` attribute is an optional convention which allows partitioning WireHose resources, channel factories, tags and other objects into groups.

First you'll model the RSSFeed's title. We're calling it "name" instead of "title" because it's a convention for WireHose resources to have a name, if possible.

1. Select the **RSSFeed** entity.
2. Choose **Add Attribute** from the **Property** menu.
3. Set its prototype to **whString255**.

> **Note:** The whString255 prototype defines a column of type CHAR(255). You can safely redefine all the whString prototypes to be VARCHAR columns of any size. See the WHEnterpriseObject reference for details.

4. Remove the padlock icon so this attribute isn't used for locking.
5. Set the attribute's name to **name**, and set its column name to **NAME**.



Next, you'll model the RSSFeed's link.

1. Choose **Add Attribute** from the **Property** menu.
2. Set its prototype to **whString255**.
3. Remove the padlock icon so this attribute isn't used for locking.
4. Set the attribute's name to **link**, and set its column name to **LINK**.

Next, you'll model the RSSFeed's description. Since "description" is a reserved word in WebObjects, it's a WireHose convention to name the attribute "textDescription".

1. Choose **Add Attribute** from the **Property** menu.
2. Set its prototype to **whString255**.
3. Remove the padlock icon so this attribute isn't used for locking.
4. Set the attribute's name to **textDescription**, and set its column name to **TEXTDESC**.

There are a number of other attributes defined in the RSS 2.0 specification which we are not modeling here for simplicity's sake. However, we will model some more attributes which will be used in the Hello World application.

The first is a "lastFetchDate" property, which is the date the items from a feed were last imported.

1. Choose **Add Attribute** from the **Property** menu.
2. Set its prototype to **whTimestamp**.
3. Remove the padlock icon so this attribute isn't used for locking.
4. Set the attribute's name to **lastFetchDate**, and set its column name to **LASTFETCH**.

Since there are several versions of RSS, some feeds may be valid but not supported by Hello World. The last attribute you'll model is a boolean indicating whether or not the feed was valid the last time we fetched, so the crawler can skip those feeds the next time.

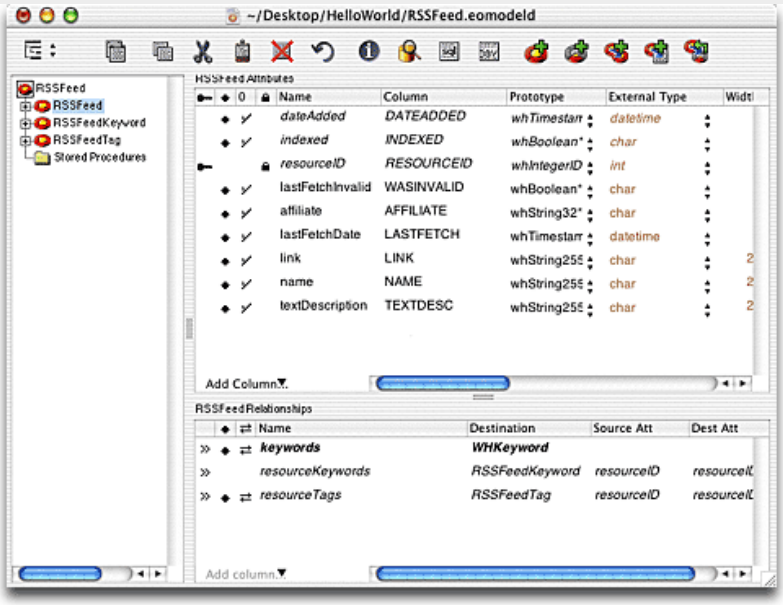
1. Choose **Add Attribute** from the **Property** menu.
2. Set its prototype to **whBoolean**.

> **Note:** There is no standard way to store booleans in a database; some support BOOLEAN columns, while others may store a "Y" or "N" in a CHAR(1) column, or a zero or one in an INTEGER. WireHose provides the whBoolean prototype and several utility methods which make

it easy to deal with varying definitions for boolean attributes in a transparent fashion.

3. Remove the padlock icon so this attribute isn't used for locking.
4. Set the attribute's name to **lastFetchInvalid**, and set its column name to **WASINVALID**.

Note: Your Java code will actually call methods named `lastFetch`**`Was`**`Invalid` and `setLastFetch`**`Was`**`Invalid`.



Uniquing items

Next you'll tell WireHose which attributes should be used to distinguish one feed from another during importing. In this case, duplicate feeds have either the same name or link.

1. Select the **RSSFeed** entity.
2. Choose **Inspector...** from the **Tools** menu.
3. Select the **EOEntity UserInfo** pane.
4. Click **Add**.
5. Set the key name to **WHKeysForUniquing**, and its value to **(link, name)**.

Modeling items

You'll use the same techniques to model the RSSItem entity as you did for RSSFeed. Since the two share some common attributes, you'll be able to copy and paste them to save some time.

Creating RSSItem

Note: The new resource template creates a new EOModel for each resource entity. You can leave each entity in a separate model file, or merge them into a single model by copying and pasting the entities. For this tutorial we'll leave them in separate models.

1. Select the **Entities** group in the Files pane in Project Builder.
2. Choose **New File...** from the **File** menu. Scroll down to the WireHose **Resource** template and click **Next**.
3. Name it **RSSItem**, add it to the Application Server target in the Hello World project, and click **Finish**.
4. Open the **RSSItem.eomodeld** file in EOModeler.
5. Choose **Set Adaptor Info...** from the **Model** menu.
6. Set the URL and driver as entered in your adaptor dictionary:

   If you're using OpenBase:

Set the URL to **jdbc:openbase://127.0.0.1/HelloWorld** and the driver to **com.openbase.jdbc.ObDriver**

If you're using FrontBase:

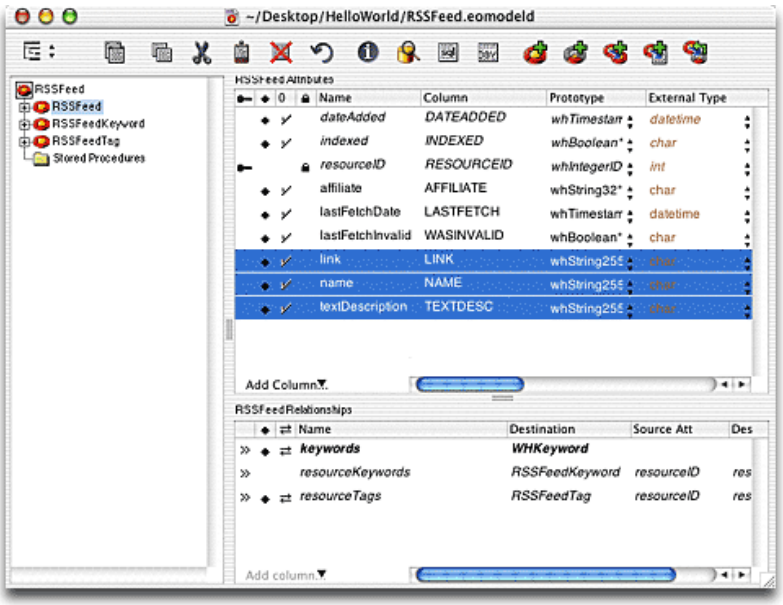
Set the URL to **jdbc:FrontBase://localhost/HelloWorld/user=HelloWorld** and the driver to **jdbc.FrontBase.FBJDriver**

7. Click **OK**.
8. Change the name of the **MyResource** entity to **RSSItem**, its table to **RSSItem**, and its class name to **RSSItem**.
9. Change the name of the **MyResourceKeyword** entity to **RSSFeedKeyword**, and its table to **RSSItemKw**.
10. Change the name of the **MyResourceTag** entity to **RSSItemTag**, and its table to **RSSItemTag**.

Adding attributes

Since RSS feeds and items both contain titles, links and descriptions, you can copy and paste the attribute definitions from the RSSFeed entity into RSSItem.

1. Switch to **RSSFeed.eomodeld**.
2. Select the **RSSFeed** entity.
3. Select the **link**, **name** and **textDescription** attributes.



4. Choose **Copy** from the **Edit** menu.
5. Switch to **RSSitem.eomodeld**.
6. Select the **RSSItem** entity.
7. Choose **Paste** from the **Edit** menu.
8. Remove the padlock icon from the **textDescription**, **link** and **name** attributes so they won't be used for locking.

Uniquing items

Next you'll tell WireHose which attributes should be used to distinguish one item from another during importing. This is important since new items will typically be added to the top of the feed, pushing older items off the bottom. If Hello World's crawler didn't have a method for determining if it had already seen an item, the database would be cluttered with duplicates.

In this case, duplicate items have either the same name or link.

1. Select the **RSSItem** entity.
2. Choose **Inspector...** from the **Tools** menu.
3. Select the **EOEntity UserInfo** pane.
4. Click **Add**.
5. Set the key name to **WHKeysForUniquing**, and its value to **(link, name)**.

Note: The RSS 2.0 specification allows feed providers to provide an optional attribute for each item, called "guid". This globally unique identifier is specifically intended help aggregators determine if an item has been seen previously, but not all feeds use it. For simplicity, we are ignoring this attribute in this tutorial.

Relating feeds and items

Now you'll model a one−to−many relationship between feeds and items, so that each feed has multiple items, and each item has a single feed.

Note: Another way to model this relationship would be to model RSSFeed as a subentity of WHTag which would implement the WHTaggable and WHIndexable interfaces. Creating resource types which can be used to tag other resources is a very powerful technique, but beyond the scope of this tutorial.

Relating items to feeds

First you'll add a "feedID" attribute to RSSItem.

1. Select the **RSSItem** entity.
2. Choose **Add Attribute** from the **Property** menu.
3. Set its prototype to **whIntegerID**.

   Note: By convention, all primary key attributes in a WireHose application use the "whIntegerID" prototype, except for WHTag, which uses "whBinaryID". Both these prototypes are defined as INTEGER columns by default.

   WireHose provides special support for using binary primary keys. WireHose will generate them for you, and encode a reference to the entity directly in the primary key itself. This can improve performance by avoiding extra fetches when resolving to−one faults against abstract entities.
4. Remove the padlock icon so this attribute isn't used for locking.
5. Remove the diamond icon so this attribute isn't a class property.
6. Check the "Allows null" column so this attribute can be null
7. Set the attribute's name to **feedID**, and set its column name to **FEEDID**.

Next, add the relationship.

1. Choose **Add Relationship** from the **Property** menu.

2. Choose **Inspector...** from the **Tools** menu.
3. Set the relationship type to **To One**.
4. Change the destination model to **RSSFeed**.

> **Note:** You may need to quit and relaunch EOModeler to see RSSFeed in the destination model popup.

5. Change the destination entity to **RSSFeed**.
6. Set the source attribute to **feedID**, and the destination attribute to **resourceID**.
7. Click **Connect**.
8. Change the relationship's name to **feed**.

Relating feeds to items

Now you'll add the inverse relationship.

1. Switch to **RSSFeed.eomodeld**.
2. Choose **Add Relationship** from the **Property** menu.
3. Choose **Inspector...** from the **Tools** menu.
4. Set the relationship type to **To Many**.
5. Change the destination model to **RSSItem**.

> **Note:** You may need to quit and relaunch EOModeler to see RSSItem in the destination model popup.

6. Change the destination entity to **RSSItem**.
7. Set the source attribute to **resourceID**, and the destination attribute to **feedID**.

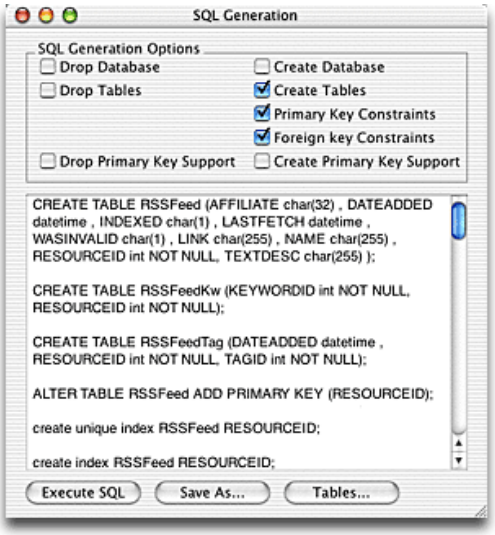
8. Click **Connect**.
9. Change the relationship's name to **items**.

Generating SQL and Java

Next you'll generate the SQL and Java for the RSSFeed and RSSItem entities. The Java classes will inherit from WHConcreteResource, which is a default implementation of the WHTaggable and WHIndexable interfaces.

Generating SQL and Java for feeds

1. Switch to **RSSFeed.eomodeld**.
2. Select the **RSSFeed**, **RSSFeedKeyword** and **RSSFeedTag** entities.
3. Choose **Generate SQL...** from the **Property** menu
4. Turn on the **Create Tables**, **Primary Key Constraints** and **Foreign Key Constraints** options, and uncheck everything else.
5. Click **Execute SQL**.



6. Close the SQL Generation window.
7. Select the **RSSFeed** entity
8. Choose **Generate Java Files...** from the **Property** menu.

9. Click **Overwrite**.
10. Open **RSSFeed.java** in Project Builder.
11. Add this line:

```
import com.wirehose.base.*;
```

12. Change the class declaration to:

```
public class RSSFeed extends com.wirehose.base.WHConcreteResource
```

13. Add this method so feeds are inserted into the database with a `lastFetchDate` in the past:

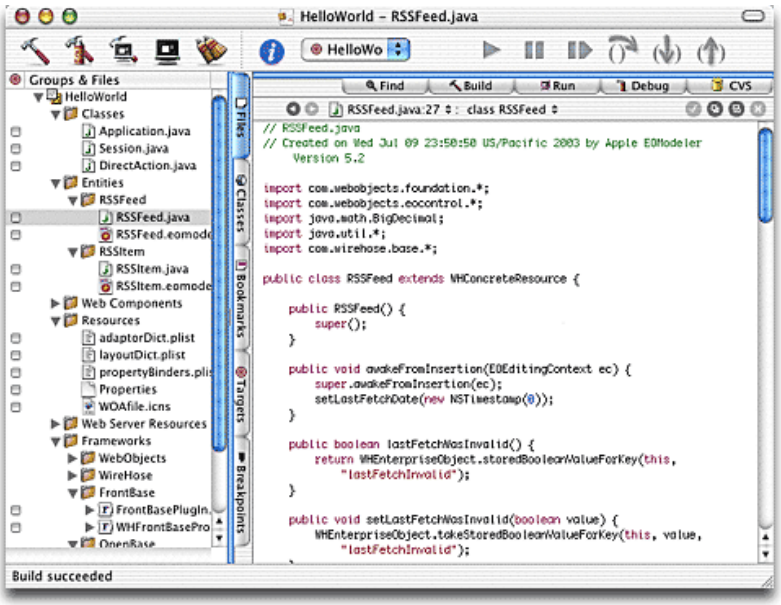
```
public void awakeFromInsertion(EOEditingContext ec) {
    super.awakeFromInsertion(ec);
    setLastFetchDate(new NSTimestamp(0));
    setLastFetchWasInvalid(false);
}
```

14. To take advantage of the boolean attribute support in WireHose, add these two methods:

```
public boolean lastFetchWasInvalid() {
    return WHEnterpriseObject.storedBooleanValueForKey(this, "lastFetchInvalid");
}

public void setLastFetchWasInvalid(boolean value) {
    WHEnterpriseObject.takeStoredBooleanValueForKey(this, value, "lastFetchInvalid");
}
```

15. Then change the `lastFetchValid` and `setLastFetchValid` methods so they accept and return instances of java.lang.Object:

```
public Object lastFetchInvalid() {
    return storedValueForKey("lastFetchInvalid");
}

public void setLastFetchInvalid(Object value) {
    takeStoredValueForKey(value, "lastFetchInvalid");
}
```



Generating SQL and Java for items

1. Switch to **RSSItem.eomodeld**.
2. Select the **RSSItem**, **RSSItemKeyword** and **RSSItemTag** entities.
3. Choose **Generate SQL...** from the **Property** menu
4. Turn on the **Create Tables**, **Primary Key Constraints** and **Foreign Key Constraints** options, and uncheck everything else.
5. Click **Execute SQL**.
6. Close the SQL Generation window.
7. Select the **RSSItem** entity
8. Choose **Generate Java Files...** from the **Property** menu.
9. Click **Overwrite**.
10. Open **RSSItem.java** in Project Builder.
11. Add this line:

```
import com.wirehose.base.*;
```

12. Change the class declaration to:

```
public class RSSItem extends com.wirehose.base.WHConcreteResource
```

Importing feeds

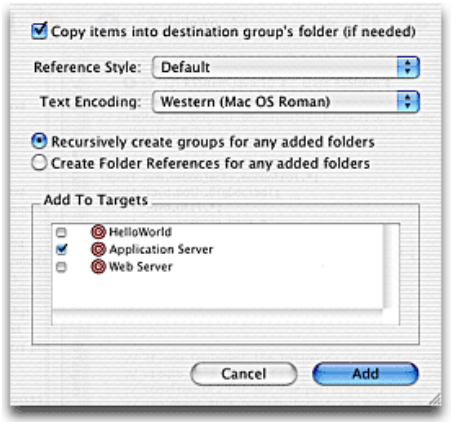
In this section, you'll write an importer which will insert information about some RSS feeds into the database.

There are a number of websites which provide information about available RSS feeds. WireHose comes with a sample list of feeds collected from Syndic8.com. The list is provided in RSS 2.0 format, so you'll be able to reuse portions of the feed importer code in the feed crawler.

Sample feeds list

The first step is to add the sample feeds list to the Hello World project.

1. Expand the **Resources** group in the Files pane.
2. Choose **Add Files...** from **Project** menu.
3. Select **SampleRSSFeeds.xml** in the **WireHoseExtras/SampleData/data/** folder, and click **Add**.
4. Check "Copy items into destination group's folder (if needed)", add it to the Application Server target, and click **Add**.



The sample feeds file looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="2.0">
<channel>
<title>WireHose Hello World RSS Feed List (from Syndic8.com)</title>
<link></link>
<description>Sample RSS feed list for WireHose Hello World tutorial</description>
<webMaster>support@bulldogbeach.com</webMaster>
<pubDate>2003-07-01</pubDate>
<buildDate>2003-07-01</buildDate>

<item>
    <title>About.com Botany</title>
    <link>http://www.growinglifestyle.com/h117/index.rss</link>
    <description>Latest articles at About.com Botany (from Growing Lifestyle).</description>
    <category>Consumer/Gardening</category>
</item>
<item>
    <title>About.com Gardening</title>
    <link>http://www.growinglifestyle.com/h106/index.rss</link>
    <description>Latest articles at About.com Gardening (from Growing Lifestyle).</description>
    <category>Consumer/Gardening</category>
</item>
...
</channel>
</rss>
```

XML mapping model

Mapping models are used to translate XML data to objects. For importing new resources into the database, the convention is to map the XML data to an array of NSDictionary objects which represent snapshots of the individual items. For more information about mapping models, see the WebObjects documentation.

1. Expand the **Resources** group in the Files pane.
2. Choose **Add Files...** from **Project** menu.
3. Select **rss20MappingModel.xml** in the **WireHoseExtras/SampleData/maps/** folder, and click **Add**.
4. Check "Copy items into destination group's folder (if needed)", add it to the Application Server target, and click **Add**.

Here's what the mapping model looks like. This mapping model specifies all the attributes in an RSS 2.0 file, including the ones Hello World is ignoring. The attributes used by Hello World are highlighted.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<model>
    <entity name="NSMutableDictionary" xmlTag="rss" unmappedTagsKey="unmappedTags">
        <property name="channel" xmlTag="channel"/>
        <property name="version" xmlTag="version"/>
    </entity>
    <entity name="NSMutableDictionary" xmlTag="channel" unmappedTagsKey="unmappedTags" >
        <property name="name" xmlTag="title"/>
        <property name="link" xmlTag="link"/>
        <property name="textDescription" xmlTag="description"/>
        <property name="language" xmlTag="language"/>
        <property name="copyright" xmlTag="copyright"/>
        <property name="managingEditor" xmlTag="managingEditor"/>
        <property name="webMaster" xmlTag="webMaster"/>
        <property name="pubDate" xmlTag="pubDate"/>
        <property name="lastBuildDate" xmlTag="lastBuildDate"/>
        <property name="tags" xmlTag="category" forceList="YES"/>
        <property name="generator" xmlTag="generator"/>
        <property name="docs" xmlTag="docs"/>
        <property name="cloud" xmlTag="cloud"/>
        <property name="ttl" xmlTag="ttl"/>
        <property name="image" xmlTag="image"/>
        <property name="rating" xmlTag="rating"/>
        <property name="textInput" xmlTag="textInput"/>
        <property name="skipHours" xmlTag="skipHours"/>
        <property name="skipDays" xmlTag="skipDays"/>
        <property name="items" xmlTag="item" forceList="YES"/>
    </entity>
    <entity name="NSMutableDictionary" xmlTag="image" unmappedTagsKey="unmappedTags">
        <property name="url" xmlTag="url"/>
        <property name="title" xmlTag="title"/>
```

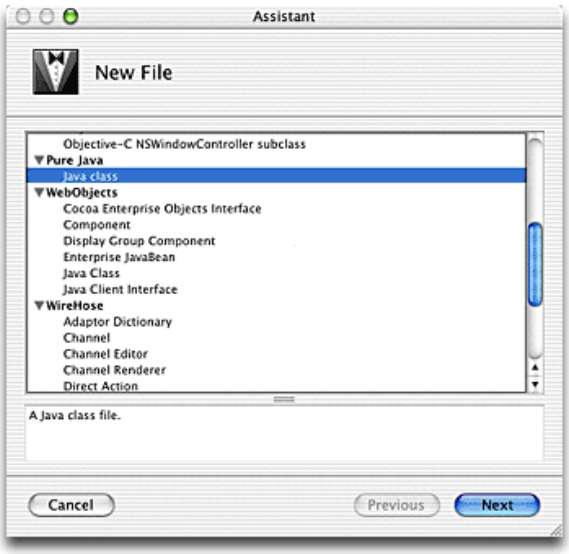
```
    <property name="link" xmlTag="link"/>
    <property name="width" xmlTag="width"/>
    <property name="height" xmlTag="height"/>
    <property name="description" xmlTag="description"/>
</entity>
<entity name="NSMutableDictionary" xmlTag="cloud" unmappedTagsKey="unmappedTags">
    <property name="domain" xmlTag="domain"/>
    <property name="port" xmlTag="port"/>
    <property name="registerProcedure" xmlTag="registerProcedure"/>
    <property name="protocol" xmlTag="protocol"/>
</entity>
<entity name="NSMutableDictionary" xmlTag="textInput" unmappedTagsKey="unmappedTags">
    <property name="title" xmlTag="title"/>
    <property name="description" xmlTag="description"/>
    <property name="name" xmlTag="name"/>
    <property name="link" xmlTag="link"/>
</entity>
<entity name="NSMutableDictionary" xmlTag="item" unmappedTagsKey="unmappedTags">
    <property name="name" xmlTag="title"/>
    <property name="link" xmlTag="link"/>
    <property name="textDescription" xmlTag="description"/>
    <property name="author" xmlTag="author"/>
    <property name="tags" xmlTag="category" forceList="YES"/>
    <property name="comments" xmlTag="comments"/>
    <property name="enclosure" xmlTag="enclosure"/>
    <property name="guid" xmlTag="guid"/>
    <property name="pubDate" xmlTag="pubDate"/>
    <property name="source" xmlTag="source"/>
</entity>
<entity name="NSMutableDictionary" xmlTag="source" unmappedTagsKey="unmappedTags" contentsKey="description">
    <property name="description" xmlTag="description"/>
```

```
        <property name="url" xmlTag="url"/>
    </entity>
    <entity name="NSMutableDictionary" xmlTag="enclosure" unmappedTagsKey="unmappedTags">
        <property name="url" xmlTag="url"/>
        <property name="length" xmlTag="length"/>
        <property name="type" xmlTag="type"/>
    </entity>
    <entity name="NSMutableDictionary" xmlTag="category" unmappedTagsKey="unmappedTags" contentsKey="id">
        <property name="id" xmlTag="id"/>
        <property name="domain" xmlTag="domain"/>
    </entity>
    <entity name="NSMutableDictionary" xmlTag="guid" unmappedTagsKey="unmappedTags" contentsKey="id">
        <property name="id" xmlTag="id"/>
        <property name="isPermaLink" xmlTag="isPermaLink"/>
    </entity>
    <entity name="NSMutableDictionary" xmlTag="enclosure" unmappedTagsKey="unmappedTags">
        <property name="title" xmlTag="title"/>
        <property name="description" xmlTag="description"/>
        <property name="name" xmlTag="name"/>
        <property name="link" xmlTag="link"/>
    </entity>
</model>
```

Fetching dictionaries

Now you'll start building Hello World's importer. This class will use the WHImporter utility class to do most of its heavy lifting.

1. Expand the **Classes** group in the Files pane.
2. Chose **New File...** from **File** menu. Scroll down to the Pure Java **Java Class** template and click **Next**.



3. Name it **Importer.java**, add it to the Application Server target in the Hello World project, and click **Finish**.

4. Add these lines to **Importer.java**

```
import com.wirehose._util.*;
import com.wirehose.base.*;
```

5. Add this method:

```
public static void importFeeds() {

    // fetch feeds list as dictionary
    NSMutableDictionary rss = WHImporter.fetchDictionaryFromURL(
      "SampleRSSFeeds.xml",
      "Contents/Resources/rss20MappingModel.xml");

    // extract feeds from dictionary and clean up tags
    NSMutableArray snapshots =
      cleanSnapshots(rss.valueForKeyPath("channel.items"));

    EOEditingContext ec = new EOEditingContext();
    ec.lock();
    try {

        // insert resources into editing context
        WHImporter.insertResources(
           ec, snapshots, "RSSFeed", "Feeds/", null,
           WHImporter.IgnoreAndTag, true, true, true, true, false);

        ec.saveChanges();
    } catch (Exception e) {
        System.out.println("Error importing resources: "+e);
        e.printStackTrace();
    }

    try {
        ec.saveChanges();
    } catch (Exception e) {
        System.out.println("Exception saving changes: "+e);
    }
    ec.unlock();
    ec.dispose();
}
```

This method uses two methods from WHImporter. The first, `fetchDictionaryFromURL,` takes two arguments which specify the location of an XML file to be imported, and the mapping model which will turn the XML into a

dictionary.

The other WHImporter method, `insertResources`, is a general−purpose utility for inserting resources into a database. It takes several arguments which specify how to handle resources which already exist in the database, whether or not to index keywords for newly inserted resources, whether to add tags to resources, etc.

The arguments used here ensure that if a new resource already exists in the database, the new resource will be ignored rather than inserted. If any tags are specified on the new resource, those tags will be assigned to the already existing resource. This handles the case where an identical resource has been imported multiple times in multiple categories, by having only a single resource with multiple tags.

The "Feeds/" argument specifies a tag path prefix. Any categories described in the feed will be appended to this string before being turned into tags. For example, if a feed has an assigned category of "Consumer/Gardening", the tag used will actually be "Feeds/Consumer/Gardening".

Cleaning snapshots

The mapping model does a fairly good job of translating the XML input into dictionaries, but it has some quirks, and WHImporter's `insertResources` expects those dictionaries to be in a particular format. The importer needs a `cleanSnapshots` method which will fix what we get from `fetchDictionaryFromURL`.

1. Add this method:

```
static NSMutableArray cleanSnapshots(Object whatWeFound) {

    NSMutableArray snapshots;

    // "forcelist" in the mapping model is unreliable
    // sometimes we get a single object, so pack it into an array
    if (whatWeFound instanceof NSArray) {
        snapshots = (NSMutableArray)whatWeFound;
    } else {
        snapshots = new NSMutableArray(whatWeFound);
    }
```

```
    NSMutableDictionary snapshot;
    NSKeyValueCoding tags;

    // iterate through snapshots
    // for each snapshot, extract tags from content key
    for (int i=0, count=snapshots.count(); i<count; i++) {
        snapshot = (NSMutableDictionary)snapshots.objectAtIndex(i);
        tags = (NSKeyValueCoding)snapshot.objectForKey("tags");
        if (tags != null) {

            // tags will be either a dictionary with one key "id"
            // mapping to a string indicating the tagpath,
            // or an array of dictionaries, each with an "id" key.

            // Calling valueForKey on an array will construct a new
            // array with the results of calling valueForKey on each
            // object in the old array... nice.

            // So we end up with either a string, or an array of strings
            snapshot.setObjectForKey(tags.valueForKey("id"), "tags");
            snapshots.replaceObjectAtIndex(snapshot, i);
        }
    }
    return snapshots;
}
```

The mapping model specifies that RSS items should be mapped to a list through its `forceList` property. Sometimes the XML importer returns a single item instead of a list, so this method will pack the object into an array.

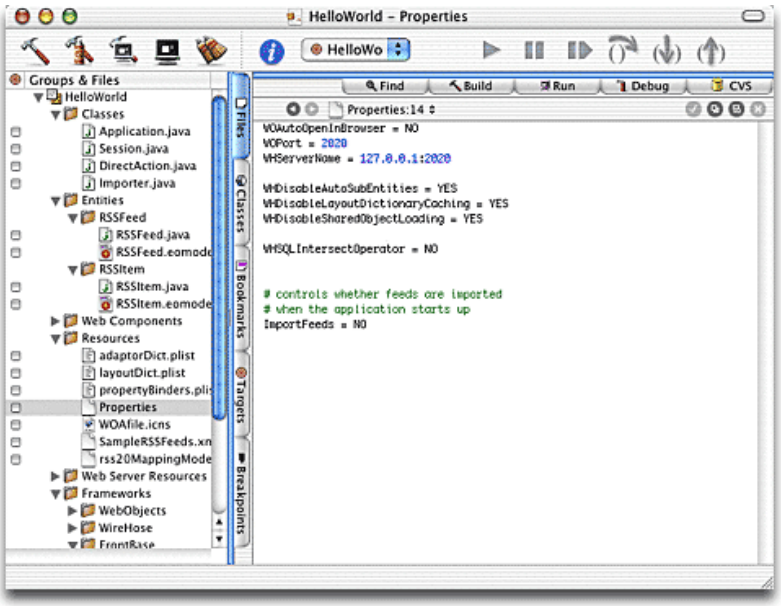
The RSS 2.0 format specifies that <category> is a container element. The `contentsKey` property in the mapping model specifies that the XML importer should map a category to a dictionary with a single key, "id", which maps to the name of the category itself. If an item has multiple category entries, then the importer will return an array of dictionaries. The `cleanSnapshots` method extracts the category (or categories) from the dictionary (or dictionaries).

Enabling the importer

Next, we'll add a property that controls whether or not to import feeds at runtime, and actually call the importer.

1. Select the **Properties** file in the Resources group in the Files pane, and add these lines:

```
# controls whether feeds are imported
# when the application starts up
ImportFeeds = NO
```



2. Select **Application.java** in the Classes group, and uncomment this line in the constructor:

```
NSNotificationCenter.defaultCenter().addObserver(
    this, new NSSelector("initialize", new Class[] { NSNotification.class } ),
    WHApplicationHelper.ApplicationHelperDidFinishInitializing, null);
```

WireHose provides an object called the WHApplicationHelper to handle application−level behavior. Among other tasks, WHApplicationHelper handles various initialization and setup tasks. Once it's done, it posts an `ApplicationHelperDidFinishInitializing` notification which indicates that it's now safe to access WireHose tags and other data structures.

3. Now add this method, which will be called in response to the notification:

```
public void initialize(NSNotification notification) {
    if (NSPropertyListSerialization.booleanForString(
        System.getProperty("ImportFeeds"))) {
        Importer.importFeeds();
    }
}
```

If the "ImportFeeds" property evaluates to true (or "YES"), then the importer will run.

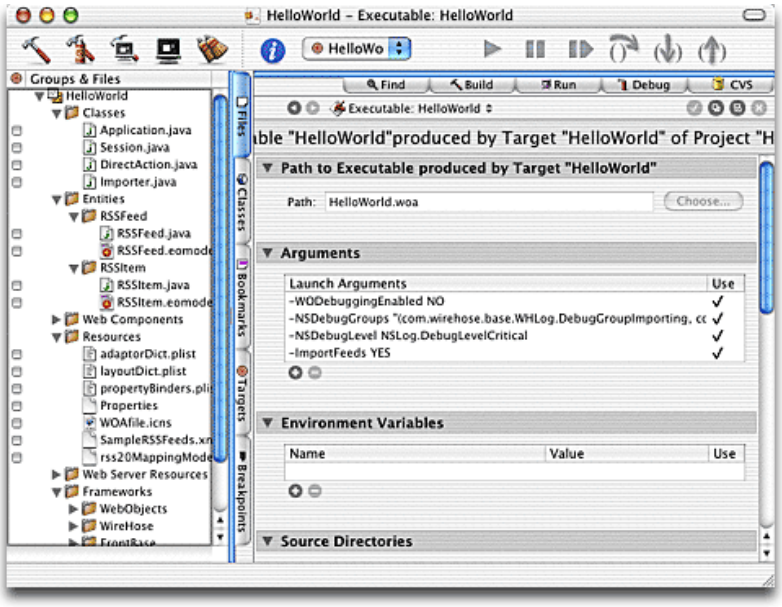
Enabling logging

WireHose uses the NSLog class to provide logging about its behavior. In this example, we want to enable logging for the WHImporter class so we can see the feeds being imported.

1. Choose **Edit Active Executable 'HelloWorld'** from the **Project** menu.
2. Click the **+** icon under **Launch Arguments** to add each of these arguments:

```
-WODebuggingEnabled NO
-NSDebugGroups "(com.wirehose.base.WHLog.DebugGroupImporting,
        com.wirehose.base.WHLog.DebugGroupWireHose)"
-NSDebugLevel NSLog.DebugLevelCritical
-ImportFeeds YES
```

3. Check the **Use** column for each of the arguments.



Running the importer

Now build and launch Hello World, and watch as the feeds are imported. You don't need to import all the feeds for this example, so you can stop the application once a few dozen have been imported. The output should look something like this:

```
Reading MacOSClassPath.txt ...
Launching HelloWorld.woa ...
java -XX:NewSize=2m -Xmx64m -Xms32m -DWORootDirectory="/System" -DWOLocalRootDirectory=""
-DWOUserDirectory="/Users/garyt/Library/BuildProducts" -DWOEnvClassPath=""
-DWOApplicationClass=Application -DWOPlatform=MacOS -Dcom.webobjects.pid=2597 -classpath
WOBootstrap.jar com.webobjects._bootstrap.WOBootstrap -WODebuggingEnabled NO -NSDebugGroups
"(com.wirehose.base.WHLog.DebugGroupImporting, com.wirehose.base.WHLog.DebugGroupWireHose)"
-NSDebugLevel NSLog.DebugLevelCritical -ImportFeeds YES
appRoot is /Users/garyt/Library/BuildProducts/HelloWorld.woa/Contents
Loading /Users/garyt/Library/BuildProducts/HelloWorld.woa/Contents/MacOS/MacOSClassPath.txt
Generated classpath:
  /Users/garyt/Library/BuildProducts/HelloWorld.woa/Contents/Resources/Java/HelloWorld.jar
  /System/Library/Frameworks/JavaFoundation.framework/Resources/Java/javafoundation.jar
  /System/Library/Frameworks/JavaEOControl.framework/Resources/Java/javaeocontrol.jar
  /System/Library/Frameworks/JavaEOAccess.framework/Resources/Java/javaeoaccess.jar
  /System/Library/Frameworks/JavaWebObjects.framework/Resources/Java/javawebobjects.jar
  /System/Library/Frameworks/JavaJDBCAdaptor.framework/Resources/Java/javajdbcadaptor.jar
  /System/Library/Frameworks/JavaWOExtensions.framework/Resources/Java/JavaWOExtensions.jar
  /System/Library/Frameworks/JavaXML.framework/Resources/Java/javaxml.jar
  /Library/Frameworks/WireHoseBase.framework/Resources/Java/WireHoseBase.jar
  /Library/Frameworks/WireHoseLayoutSupport.framework/Resources/Java/WireHoseLayoutSupport.jar
  /Library/Frameworks/WireHoseWOBuilderBindings.framework/
  /Library/Frameworks/WHOpenBasePrototypes.framework/
  /Library/Frameworks/OpenBasePKPlugIn.framework/Resources/Java/OpenBasePKPlugIn.jar
  /Users/garyt/Library/Java/
```

```
  /Library/Java/
  /System/Library/Java/
  /Network/Library/Java
  /Library/WebObjects/Extensions/activation.jar
  /Library/WebObjects/Extensions/avalon-framework-4.1.2.jar
  /Library/WebObjects/Extensions/axis-ant.jar
  /Library/WebObjects/Extensions/axis.jar
  /Library/WebObjects/Extensions/commons-discovery.jar
  /Library/WebObjects/Extensions/commons-logging.jar
  /Library/WebObjects/Extensions/jaxrpc.jar
  /Library/WebObjects/Extensions/log4j-1.2.4.jar
  /Library/WebObjects/Extensions/logkit-1.0.1.jar
  /Library/WebObjects/Extensions/mail.jar
  /Library/WebObjects/Extensions/saaj.jar
  /Library/WebObjects/Extensions/wsdl4j.jar
  /Library/WebObjects/Extensions/xmlrpc-1.1.jar
  /Library/WebObjects/Extensions/
[2003-07-10 00:31:45 PDT] <main> WireHose Server 3.0 -- The WireHose frameworks are
copyright 2000-2003 Bulldog Beach Interactive, Inc. All rights reserved. WireHose is a
trademark of Bulldog Beach Interactive, Inc.
[2003-07-10 00:31:46 PDT] <main> Created adaptor of class WODefaultAdaptor on port 2020 and
address icecube.bulldogbeach.com/192.168.0.101 with WOWorkerThread minimum of 16 and maximum
of 256
[2003-07-10 00:31:48 PDT] <main> Application project found: Will locate resources in
'/Users/garyt/Desktop/HelloWorld' rather than
```

```
'/Users/garyt/Library/BuildProducts/HelloWorld.woa' .
[2003-07-10 00:31:52 PDT] <main> Creating LifebeatThread now with: HelloWorld 2020
icecube.bulldogbeach.com/192.168.0.101 1085 30000
[2003-07-10 00:31:52 PDT] <main> Welcome to HelloWorld, another top-quality application
using the WireHose frameworks from Bulldog Beach Interactive. The WireHose frameworks are
Copyright 2000-2003 Bulldog Beach Interactive, Inc. All rights reserved. WireHose is a
trademark of Bulldog Beach Interactive, Inc.
[2003-07-10 00:31:52 PDT] <main> The WireHose-specific defaults are:
    WHAdaptorDict = adaptorDict.plist
    WHComponentsWithContentAreStateless = YES
    WHCookieDomain = default
    WHCookiePath = /
    WHDefaultAffiliate = default
    WHDefaultLayout = Default
    WHDefaultTagEntity = WHTag
    WHDisableAutoSubEntities = YES
    WHDisableGuestPreloading = NO
    WHDisableLayoutDictionaryCaching = YES
    WHDisableSharedObjectLoading = YES
    WHHeaderDebugEnabled = NO
    WHIgnoreMissingEntities = YES
    WHLayoutDict = layoutDict.plist
    WHLookupDictionaryDebugEnabled = NO
    WHRewriteSessionCookiePath = YES
    WHSQLExceptOperator = EXCEPT
```

```
    WHSQLIntersectOperator = NO
    WHSQLTimestampFormat = default
    WHServerName = 127.0.0.1:2020
    WHServerNameHeaderKeys = ( "x-webobjects-server-name", "SERVER_NAME", "WHServerName" )
    WHStopWordsList = stopwords.txt
    WHTagCacheSize = 1024
    WHUseEntityHints = YES
    WHUserAgentHeaderKeys = ( "HTTP_USER_AGENT", "user-agent" )
    WHUserEntityName = WHUser
[2003-07-10 00:31:53 PDT] <main> WHDisableSharedObjectLoading=YES, disabled shared object
loading
[2003-07-10 00:31:53 PDT] <main> EOModel 'RSSFeed' loaded... Connection dictionary replaced.
[2003-07-10 00:31:54 PDT] <main> EOModel 'RSSItem' loaded... Connection dictionary replaced.
[2003-07-10 00:31:54 PDT] <main> EOModel 'WireHoseBase' loaded... Connection dictionary
replaced, URL was 'jdbc:FrontBase://localhost/wirehose/user=wirehose', is now:
'jdbc:openbase://127.0.0.1/HelloWorld'.
[2003-07-10 00:31:54 PDT] <main> EOModel 'WHOpenBasePrototypes' loaded... Didn't find
WHShouldReplaceAdaptorDictionary=YES in userInfo, will not replace adaptor dictionary.
[2003-07-10 00:32:04 PDT] <main> Importing [RSSFeed 3d7457] 2003-07-10 07:32:04 Etc/GMT
About.com Botany...
[2003-07-10 00:32:05 PDT] <main> WireHose frameworks: Found valid license key. Unlimited
transactions per minute. Non-expiring.
[2003-07-10 00:32:07 PDT] <main> [Adding tags to 1... ]
[2003-07-10 00:32:07 PDT] <main> Importing [RSSFeed eef0a8] 2003-07-10 07:32:08 Etc/GMT
About.com Gardening...
```

```
[2003-07-10 00:32:09 PDT] <main> Importing [RSSFeed a16977] 2003-07-10 07:32:09 Etc/GMT
About.com Home Repair...
[2003-07-10 00:32:09 PDT] <main> Importing [RSSFeed b25572] 2003-07-10 07:32:10 Etc/GMT
About.com Interactive Fiction...
[2003-07-10 00:32:11 PDT] <main> Importing [RSSFeed b2311b] 2003-07-10 07:32:11 Etc/GMT
About.com Interior Decorating...
[2003-07-10 00:32:11 PDT] <main> Importing [RSSFeed 7f7fe] 2003-07-10 07:32:12 Etc/GMT
About.com Landscaping...
[2003-07-10 00:32:13 PDT] <main> Importing [RSSFeed 557211] 2003-07-10 07:32:13 Etc/GMT
About.com Publishing...
[2003-07-10 00:32:16 PDT] <main> Importing [RSSFeed 7fa3f6] 2003-07-10 07:32:16 Etc/GMT
About.com Roses...
[2003-07-10 00:32:16 PDT] <main> Importing [RSSFeed 1167f3] 2003-07-10 07:32:17 Etc/GMT
About.com Woodworking...
[2003-07-10 00:32:18 PDT] <main> Importing [RSSFeed d9edbe] 2003-07-10 07:32:18 Etc/GMT
Absolute Quake Files Archive...
```

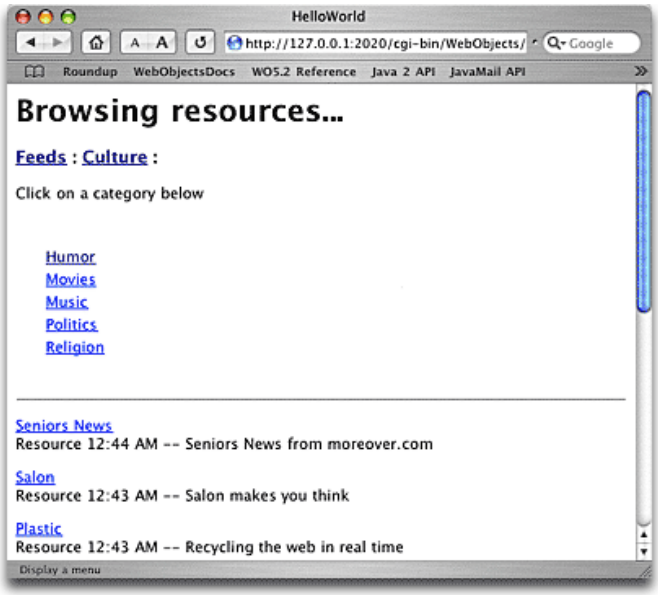
Browsing feeds

Now that you've imported some feeds, it's time to view them in the web browser. The ability to browse through tags and view available resources is built into WireHose.

1. Choose **Edit Active Executable 'HelloWorld'** from the **Project** menu.
2. Uncheck the **Use** column for the `-ImportFeeds YES` argument so feeds won't be imported.
3. Launch the application, and open this URL in your browser:

   ```
   http://127.0.0.1:2020/cgi-bin/WebObjects/HelloWorld.woa/wa/Drill
   ```
4. Browse through the available categories to see the feeds.

   The Drill direct action, defined in the WireHoseLayoutSupport framework, acts as a cover for the WHTagDrillerPage component. A tag driller page renders the current tag and its child tags, and displays the resources tagged with the current tag. Each tag is rendered as a hypertext link to the Drill direct action with its path.


HelloWorld
http://127.0.0.1:2020/cgi-bin/WebObjects/
Google
Roundup
WebObjectsDocs
WO5.2 Reference
Java 2 API
JavaMail API
Browsing resources...
Feeds : Culture :
Click on a category below
Humor
Movies
Music
Politics
Religion
Seniors News
Resource 12:44 AM -- Seniors News from moreover.com
Salon
Resource 12:43 AM -- Salon makes you think
Plastic
Resource 12:43 AM -- Recycling the web in real time
Display a menu

Crawling feeds

Importing RSS items into the database is similar to importing feeds, except that as an aggregator, Hello World will need to crawl the feeds periodically to fetch any updated items. The crawler will ignore any feeds which had problems the last time they were fetched. It will also tag each item as it is fetched so users can browse through items by category.

Fetching feeds to crawl

First, a method which fetches the set of feeds to crawl.

1. Add this method to **Importer.java**:

```
static NSArray fetchFeedsToCrawl(EOEditingContext ec) {
    NSMutableArray qualifiers = new NSMutableArray();

    // fetch all feeds that haven't been fetched in the last hour
    qualifiers.addObject(new EOKeyValueQualifier(
       "lastFetchDate",
        EOQualifier.QualifierOperatorLessThan,
        new NSTimestamp().timestampByAddingGregorianUnits(0, 0, 0, -1, 0, 0)));

    // and all feeds which weren't invalid last time we fetched
    qualifiers.addObject(
        WHEnterpriseObject.qualifierForBooleanAttribute(ec,
        "RSSFeed", "lastFetchInvalid", EOQualifier.QualifierOperatorEqual, false));

    EOQualifier q = new EOAndQualifier(qualifiers);
    EOFetchSpecification fs = new EOFetchSpecification("RSSFeed", q, null);
    return ec.objectsWithFetchSpecification(fs);
}
```

> **Note:** The `qualifierForBooleanAttribute` method constructs a qualifier to match boolean values in a database independent fashion. It works by inspecting the current definition of the boolean attribute to determine whether to use a Boolean, String or Integer to represent true or false. Note that here we are using "lastFetchInvalid", which is what the attribute is called in the model. When setting or getting a boolean value on the feed, you'll use the `setLastFetch`**`Was`**`Invalid` and `lastFetch`**`Was`**`Invalid` methods defined earlier.

Crawling feeds

The next step is to write the crawler. This method is similar to the `importFeeds` method, except that it will call `fetchDictionaryFromURL` and `insertResources` repeatedly, once for each available feed. It will also assign tags to the items in the feed based on the feed's tags.

1. Add this method to **Importer.java**:

```
public static void crawlFeeds() {
    EOEditingContext ec = new EOEditingContext();
    ec.lock();

    NSArray feeds = fetchFeedsToCrawl(ec);
    NSLog.debug.appendln("Found "+feeds.count()+" to crawl...");

    RSSFeed feed;
    NSMutableDictionary rss;
    NSMutableArray snapshots;
    NSDictionary statusDict;
    NSArray inserted;

    // iterate through feeds and fetch items from each one
    for (int i=0, count=feeds.count(); i<count; i++) {
        feed = (RSSFeed)feeds.objectAtIndex(i);
        NSLog.debug.appendln("Crawling "+feed.name()+": "+feed.link());

        try {

            // import the dictionary from the feed's URL
            rss = WHImporter.fetchDictionaryFromURL(
              feed.link(), "Contents/Resources/rss20MappingModel.xml");

            // extract and clean up the dictionaries
            snapshots = cleanSnapshots(rss.valueForKeyPath("channel.items"));

            // insert the resources into the database
            // insertResources returns a dictionary of
            // inserted, updated, deleted items
            statusDict = WHImporter.insertResources(ec,
                snapshots, "RSSItem", "Content/", null,
```

```
                    WHImporter.IgnoreAndTag,
                    true, true, true, true, false);

                // get inserted items from the returned dictionary
                inserted = (NSArray)statusDict.objectForKey(WHImporter.InsertedKey);

                // add tags to the inserted items based on the feed's tags
                tagItemsForFeed(ec, inserted, feed);

                // don't fetch for another hour
                feed.setLastFetchDate(new NSTimestamp());

                ec.saveChanges();
            } catch (Exception e) {
                NSLog.debug.appendln("Exception importing "+feed.link()+" - "+e);
                feed.setLastFetchWasInvalid(true);
            }
        }
        ec.unlock();
        ec.dispose();
    }
```

The `insertResources` method returns a status dictionary which contains arrays of updated, inserted, removed and ignored objects. The `importFeeds` method ignored this return value, but here the list of inserted items are extracted from the dictionary so they can be tagged.

Tagging items

In addition to using whatever categories were specified for an item in its RSS feed, the Hello World crawler will also assign tags based on the feed's categories.

1. Add this method to **Importer.java**:

```
static void tagItemsForFeed(EOEditingContext ec, NSArray items, RSSFeed feed) {

    // use this to get a tagpath without "Feeds/" at the beginning
    // for example, "Consumer/Gardening" instead of "Feeds/Consumer/Gardening"
    WHTag feedAncestor = WHTag.tagForPath(ec, "Feeds", false);
    NSMutableArray tags = new NSMutableArray();
```

```
        String path;
        WHTag tag;

        // iterate through the feed's tags
        // and build an array of tags to assign to items
        for (int i=0, count=feed.tags().count(); i<count; i++) {
            tag = (WHTag)feed.tags().objectAtIndex(i);

            // get a tagpath that starts with "Content/" instead of "Feeds/"
            // e.g., "Content/Consumer/Gardening"
            path = "Content/"+tag.tagPath(feedAncestor, "/");

            // add a tag for that path
            tags.addObject(WHTag.tagForPath(ec, path, true));

            // and add a tag for that path, plus the feed's name
            // e.g., "Consumer/Gardening/About.com Botany"
            tags.addObject(WHTag.tagForPath(ec, path+"/"+feed.name(), true));
        }

        RSSItem item;

        // iterate through items
        for (int i=0, count=items.count(); i<count; i++) {
            item = (RSSItem)items.objectAtIndex(i);

            // associate the item with the feed
            item.addObjectToBothSidesOfRelationshipWithKey(feed, "feed");

            // and add tags to the item
            WHTag.addTags(item, tags);
        }
    }
```

Running the import

Next, you'll add another system property to control whether or not feeds get crawled at runtime. Then it's time to test the crawler.

1. Add these lines to the **`initialize`** method in **Application.java**:

   ```
   if (NSPropertyListSerialization.booleanForString(
       System.getProperty("CrawlFeeds"))) {
       Importer.crawlFeeds();
   }
   ```

   Add these lines to the **Properties** file:

   ```
   # controls whether feeds are crawled at runtime
   CrawlFeeds = NO
   ```

2. Choose **Edit Active Executable 'HelloWorld'** from the **Project** menu.
3. Click the **+** icon under **Launch Arguments** to add this argument, and check the **Use** column:

   ```
   -CrawlFeeds YES
   ```

Now build and launch Hello World, and watch as the feeds are crawled. You don't need to crawl all the feeds for this example, so you can stop the application once a few have been imported. The output should look something like this:

```
Reading MacOSClassPath.txt ...
Launching HelloWorld.woa ...
java −XX:NewSize=2m −Xmx64m −Xms32m −DWORootDirectory="/System" −DWOLocalRootDirectory=""
−DWOUserDirectory="/Users/garyt/Library/BuildProducts" −DWOEnvClassPath=""
−DWOApplicationClass=Application −DWOPlatform=MacOS −Dcom.webobjects.pid=6807 −classpath
WOBootstrap.jar com.webobjects._bootstrap.WOBootstrap −WODebuggingEnabled NO −NSDebugGroups
"(com.wirehose.base.WHLog.DebugGroupImporting, com.wirehose.base.WHLog.DebugGroupWireHose)"
−NSDebugLevel NSLog.DebugLevelCritical −CrawlFeeds YES
appRoot is /Users/garyt/Library/BuildProducts/HelloWorld.woa/Contents
Loading /Users/garyt/Library/BuildProducts/HelloWorld.woa/Contents/MacOS/MacOSClassPath.txt
Generated classpath:
  /Users/garyt/Library/BuildProducts/HelloWorld.woa/Contents/Resources/Java/HelloWorld.jar
  /System/Library/Frameworks/JavaFoundation.framework/Resources/Java/javafoundation.jar
  /System/Library/Frameworks/JavaEOControl.framework/Resources/Java/javaeocontrol.jar
  /System/Library/Frameworks/JavaEOAccess.framework/Resources/Java/javaeoaccess.jar
  /System/Library/Frameworks/JavaWebObjects.framework/Resources/Java/javawebobjects.jar
  /System/Library/Frameworks/JavaJDBCAdaptor.framework/Resources/Java/javajdbcadaptor.jar
  /System/Library/Frameworks/JavaWOExtensions.framework/Resources/Java/JavaWOExtensions.jar
  /System/Library/Frameworks/JavaXML.framework/Resources/Java/javaxml.jar
  /Library/Frameworks/WireHoseBase.framework/Resources/Java/WireHoseBase.jar
  /Library/Frameworks/WireHoseLayoutSupport.framework/Resources/Java/WireHoseLayoutSupport.jar
  /Library/Frameworks/WireHoseWOBuilderBindings.framework/
  /Library/Frameworks/WHOpenBasePrototypes.framework/
  /Library/Frameworks/OpenBasePKPlugIn.framework/Resources/Java/OpenBasePKPlugIn.jar
  /Users/garyt/Library/Java/
```

```
  /Library/Java/
  /System/Library/Java/
  /Network/Library/Java
  /Library/WebObjects/Extensions/activation.jar
  /Library/WebObjects/Extensions/avalon-framework-4.1.2.jar
  /Library/WebObjects/Extensions/axis-ant.jar
  /Library/WebObjects/Extensions/axis.jar
  /Library/WebObjects/Extensions/commons-discovery.jar
  /Library/WebObjects/Extensions/commons-logging.jar
  /Library/WebObjects/Extensions/jaxrpc.jar
  /Library/WebObjects/Extensions/log4j-1.2.4.jar
  /Library/WebObjects/Extensions/logkit-1.0.1.jar
  /Library/WebObjects/Extensions/mail.jar
  /Library/WebObjects/Extensions/saaj.jar
  /Library/WebObjects/Extensions/wsdl4j.jar
  /Library/WebObjects/Extensions/xmlrpc-1.1.jar
  /Library/WebObjects/Extensions/
[2003-07-10 02:50:46 PDT] <main> WireHose Server 3.0 -- The WireHose frameworks are
copyright 2000-2003 Bulldog Beach Interactive, Inc. All rights reserved. WireHose is a
trademark of Bulldog Beach Interactive, Inc.
[2003-07-10 02:50:48 PDT] <main> Created adaptor of class WODefaultAdaptor on port 2020 and
address icecube.bulldogbeach.com/192.168.0.101 with WOWorkerThread minimum of 16 and maximum
of 256
[2003-07-10 02:50:49 PDT] <main> Application project found: Will locate resources in
'/Users/garyt/Desktop/HelloWorld' rather than
```

```
'/Users/garyt/Library/BuildProducts/HelloWorld.woa' .
[2003-07-10 02:50:55 PDT] <main> Creating LifebeatThread now with: HelloWorld 2020
icecube.bulldogbeach.com/192.168.0.101 1085 30000
[2003-07-10 02:50:55 PDT] <main> Welcome to HelloWorld, another top-quality application
using the WireHose frameworks from Bulldog Beach Interactive. The WireHose frameworks are
Copyright 2000-2003 Bulldog Beach Interactive, Inc. All rights reserved. WireHose is a
trademark of Bulldog Beach Interactive, Inc.
[2003-07-10 02:50:55 PDT] <main> The WireHose-specific defaults are:
    WHAdaptorDict = adaptorDict.plist
    WHComponentsWithContentAreStateless = YES
    WHCookieDomain = default
    WHCookiePath = /
    WHDefaultAffiliate = default
    WHDefaultLayout = Default
    WHDefaultTagEntity = WHTag
    WHDisableAutoSubEntities = YES
    WHDisableGuestPreloading = NO
    WHDisableLayoutDictionaryCaching = YES
    WHDisableSharedObjectLoading = YES
    WHHeaderDebugEnabled = NO
    WHIgnoreMissingEntities = YES
    WHLayoutDict = layoutDict.plist
    WHLookupDictionaryDebugEnabled = NO
    WHRewriteSessionCookiePath = YES
    WHSQLExceptOperator = EXCEPT
```

```
    WHSQLIntersectOperator = NO
    WHSQLTimestampFormat = default
    WHServerName = 127.0.0.1:2020
    WHServerNameHeaderKeys = ( "x-webobjects-server-name", "SERVER_NAME", "WHServerName" )
    WHStopWordsList = stopwords.txt
    WHTagCacheSize = 1024
    WHUseEntityHints = YES
    WHUserAgentHeaderKeys = ( "HTTP_USER_AGENT", "user-agent" )
    WHUserEntityName = WHUser
[2003-07-10 02:50:56 PDT] <main> WHDisableSharedObjectLoading=YES, disabled shared object
loading
[2003-07-10 02:50:58 PDT] <main> EOModel 'RSSFeed' loaded... Connection dictionary replaced.
[2003-07-10 02:50:58 PDT] <main> EOModel 'RSSItem' loaded... Connection dictionary replaced.
[2003-07-10 02:50:58 PDT] <main> EOModel 'WireHoseBase' loaded... Connection dictionary
replaced, URL was 'jdbc:FrontBase://localhost/wirehose/user=wirehose', is now:
'jdbc:openbase://127.0.0.1/HelloWorld'.
[2003-07-10 02:50:58 PDT] <main> EOModel 'WHOpenBasePrototypes' loaded... Didn't find
WHShouldReplaceAdaptorDictionary=YES in userInfo, will not replace adaptor dictionary.
[2003-07-10 02:51:09 PDT] <main> Found 713 to crawl...
[2003-07-10 02:51:09 PDT] <main> Crawling About.com Botany:
http://www.growinglifestyle.com/h117/index.rss
[2003-07-10 02:51:57 PDT] <main> Importing [RSSItem 48854d] 2003-07-10 09:51:57 Etc/GMT
Concrete Countertops: Design, Form, and Finishes for the ......
[2003-07-10 02:52:00 PDT] <main> WireHose frameworks: Found valid license key. Unlimited
transactions per minute. Non-expiring.
```

```
[2003-07-10 02:52:02 PDT] <main> Crawling About.com Home Repair:
http://www.growinglifestyle.com/h108/index.rss
[2003-07-10 02:52:04 PDT] <main> Importing [RSSItem ab5e0b] 2003-07-10 09:52:04 Etc/GMT
Mosquito Trap, 3/4 Acre Mosquito Catcher...
[2003-07-10 02:52:07 PDT] <main> Crawling About.com Interactive Fiction:
http://interactfiction.about.com/library/news/ifnews.rss
[Fatal Error] :34:12: Open quote is expected for attribute "NAME".
[2003-07-10 02:52:09 PDT] <main> WHImporter.fetchSnapshotsFromURL() - Error decoding root
dictionary: Open quote is expected for attribute "NAME".
[2003-07-10 02:52:09 PDT] <main> Exception importing
http://interactfiction.about.com/library/news/ifnews.rss - :
com.webobjects.appserver.xml.WOXMLException [org.xml.sax.SAXParseException] Open quote is
expected for attribute "NAME".
[2003-07-10 02:52:09 PDT] <main> Crawling About.com Interior Decorating:
http://www.growinglifestyle.com/h113/index.rss
[2003-07-10 02:52:11 PDT] <main> Importing [RSSItem 24b943] 2003-07-10 09:52:11 Etc/GMT
Brill Luxus 38 Reel Push Manual Mower...
[2003-07-10 02:52:13 PDT] <main> Crawling About.com Landscaping:
http://www.growinglifestyle.com/h110/index.rss
[2003-07-10 02:52:15 PDT] <main> Importing [RSSItem daa156] 2003-07-10 09:52:15 Etc/GMT
Concrete Countertops: Design, Form, and Finishes for the ......
[2003-07-10 02:52:15 PDT] <main> [Adding tags to 1... ]
[2003-07-10 02:52:15 PDT] <main> Crawling About.com Roses:
http://www.growinglifestyle.com/h101/index.rss
[2003-07-10 02:52:16 PDT] <main> Importing [RSSItem 82fd0f] 2003-07-10 09:52:16 Etc/GMT
```

```
Plants of the Metroplex...
[2003-07-10 02:52:19 PDT] <main> Crawling Advogato: http://www.advogato.org/rss/articles.xml
[2003-07-10 02:52:20 PDT] <main> Importing [RSSItem 8dea20] 2003-07-10 09:52:20 Etc/GMT
White Box Vs Black Box Voting Systems...
[2003-07-10 02:52:20 PDT] <main> Importing [RSSItem 30b6a4] 2003-07-10 09:52:21 Etc/GMT Open
Advogato?...
[2003-07-10 02:52:22 PDT] <main> Importing [RSSItem d6ea02] 2003-07-10 09:52:22 Etc/GMT
Which License for Free Documentation?...
[2003-07-10 02:52:22 PDT] <main> Importing [RSSItem c1902d] 2003-07-10 09:52:23 Etc/GMT
Forking the good fork...
[2003-07-10 02:52:24 PDT] <main> Importing [RSSItem 7fa3f6] 2003-07-10 09:52:24 Etc/GMT Open
Investment...
[2003-07-10 02:52:24 PDT] <main> Importing [RSSItem c8092a] 2003-07-10 09:52:25 Etc/GMT
CounterfeitProof...
[2003-07-10 02:52:26 PDT] <main> Importing [RSSItem 4d75ae] 2003-07-10 09:52:26 Etc/GMT Open
source software and ethics...
[2003-07-10 02:52:26 PDT] <main> Importing [RSSItem 76358a] 2003-07-10 09:52:27 Etc/GMT
UKUUG Linux 2003 conference: Early Bird registration until end June...
[2003-07-10 02:52:28 PDT] <main> Importing [RSSItem c126b3] 2003-07-10 09:52:28 Etc/GMT Nine
days before Software Patent in Europe....
[2003-07-10 02:52:30 PDT] <main> Importing [RSSItem a04cf8] 2003-07-10 09:52:30 Etc/GMT How
should we encourage donations for software?...
[2003-07-10 02:52:31 PDT] <main> Crawling Aerospace and Defense Industry News:
http://www.moreover.com/cgi-local/page?o=rss&c=Aerospace%20and%20defense%20industry%20news
[2003-07-10 02:52:34 PDT] <main> Importing [RSSItem 4651f2] 2003-07-10 09:52:34 Etc/GMT Farm
```

```
machine helps Boeing production...
```

Importing in a separate thread

Since crawling the feeds should happen repeatedly while the application is running −− and we don't want to delay application startup while the feeds are crawled −− the feed crawler should run in its own thread.

1. Add this method to **Importer.java**:

```
public static void crawlFeedsInThread() {
    Thread crawler = new Thread() {
        public void run() {
            try {
                sleep(1000 * 15); // wait 15 secs before first crawl
                while (true) {
                    crawlFeeds();
                    sleep(1000 * 60 * 5); // crawl every 5 minutes
                }
            } catch (InterruptedException e) {
                System.out.println("crawler: "+e);
            }
        }
    };
    crawler.start();
}
```

2. And change this line in **Application.java**:

```
Importer.crawlFeeds();
```

   to this:

```
Importer.crawlFeedsInThread();
```

Now, when you launch Hello World, application startup isn't delayed, and the application will check every five minutes for feeds which haven't been crawled in the last hour to fetch.

> **Note:** Since EOF is not by default multithreaded, this technique will still lock the EOF stack for each fetch

or commit by the importer to the database. For maximum multithreadedness, you can create a new EOF stack for the importer. To do this, you create a new object store coordinator for the importer, and use it as the root object store for the importer's editing contexts.

1. Add this line to **Importer.java**:

   ```
   static EOObjectStoreCoordinator objStoreCoord = new
   EOObjectStoreCoordinator();
   ```

2. And change all EOEditingContext constructors in Importer.java from this:

   ```
   new EOEditingContext()
   ```

   to this:

   ```
   new EOEditingContext(objStoreCoord)
   ```

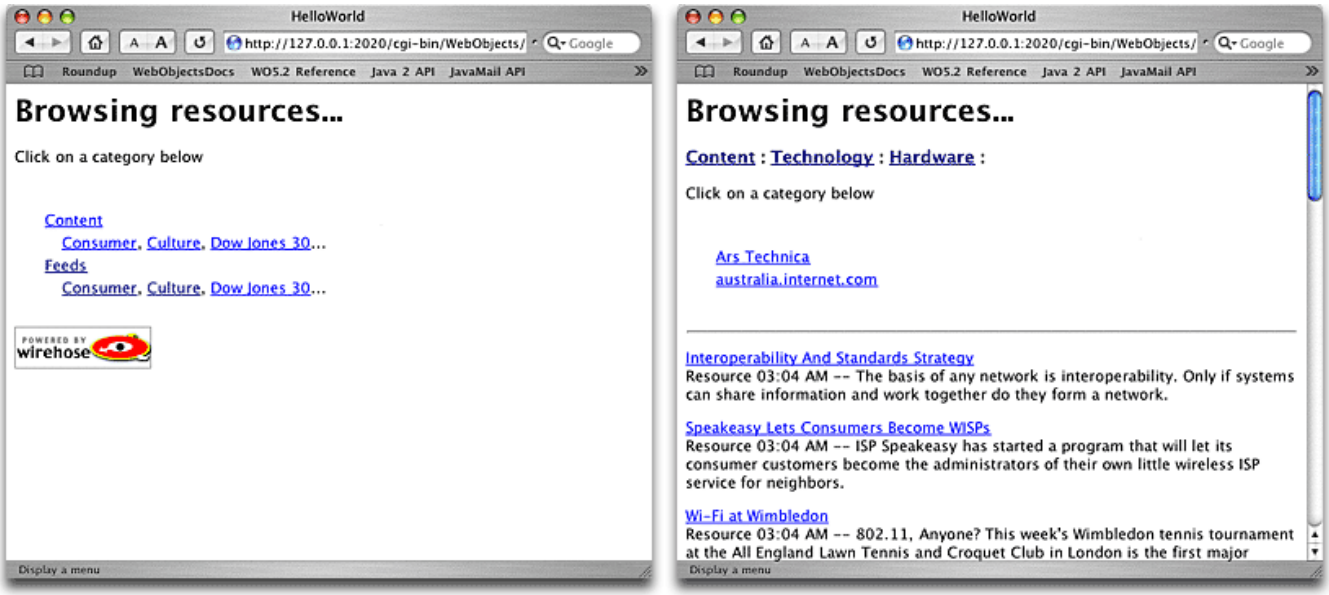
WireHose posts a `ShouldInvalidateCache` notification when new items are tagged, so objects in other EOF stacks can keep their caches up to date. See the reference documentation for WHFetcher, WHCachingDataSource, WHTagFetcher and WHTagDataSource for details. If you are deploying multiple instances of your application, you can listen for this notification and propagate it to the other instances to ensure that all the instances stay up to date.

Browsing items

If you open this URL in your browser,

```
http://127.0.0.1:2020/cgi-bin/WebObjects/HelloWorld.woa/wa/Drill
```

you can browse the items which have been imported.



Each resource is rendered using the default WHShowResource component. In the next section you'll learn how to build a custom renderer component for RSSItem objects, and enable your component in the layout dictionary.

Customizing how items are shown

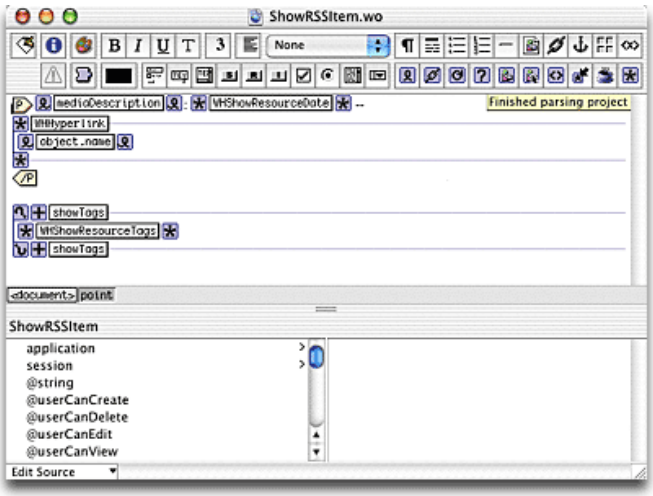
Since WireHose is an object−oriented system, with a strong emphasis on code reuse, there is a clean separation between business logic and presentation components. For maximum flexibility, WireHose allows you to use any component to render or edit an object on a page. Special components called "switchers" keep WireHose updated as to which object is currently being rendered or edited.

1. Select the **Web Components** group in the Files pane.
2. Choose **New File...** from the **File** menu. Scroll down to the WireHose **Resource Renderer** template and click **Next**.
3. Name it **ShowRSSItem**, add it to the Application Server target in the Hello World project, and click **Finish**.
4. Add this method to **ShowRSSItem.java**:

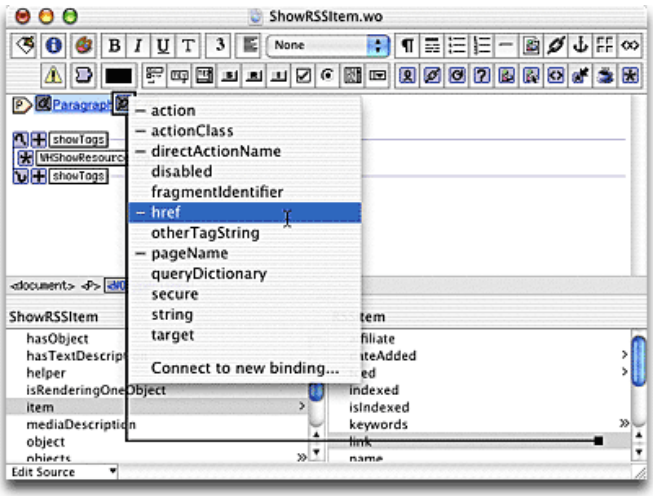
```
public RSSItem item() {
    return (RSSItem)object();
}
```

> **Note:** This method isn't strictly required; it's a convenience so that WebObjects Builder will display the bindings available for RSSItem objects. You can bind values such as the item's name to either **object.name** or **item.name**.

5. Open **ShowRSSItem.wo** in WebObjects Builder.



6. Delete the paragraph and insert a new one. Inside it, insert a WOHyperlink, and bind its **href** to **item.link**



7. Inside the link, insert a WOString. Bind its **value** to **item.name**
8. Insert a couple dashes, and another WOString. Bind its **value** to **item.textDescription**

> **Note:** Since "title" is an optional attribute in an RSS item, you may want to use WOConditional components to render the link differently depending on whether or not the item has a name.

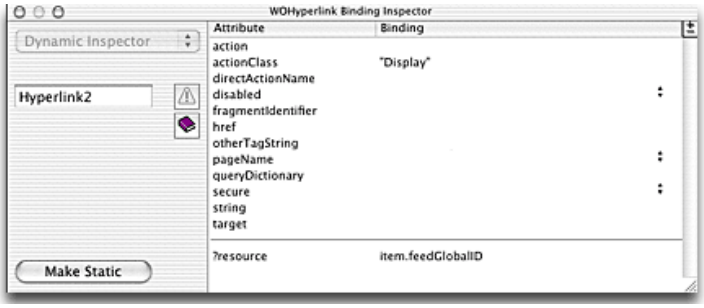
Since it's traditional in an aggregator to indicate where an item originated, you can add a link

to the item's feed. This link will use the WireHose "Display" direct action, which acts as a cover for the WHShowObjectPage component. The Display direct action takes one parameter, "resource", set to the global ID of the object to be displayed. WHEnterpriseObject provides utility methods to encode and decode globalIDs as compact strings suitable for this purpose.

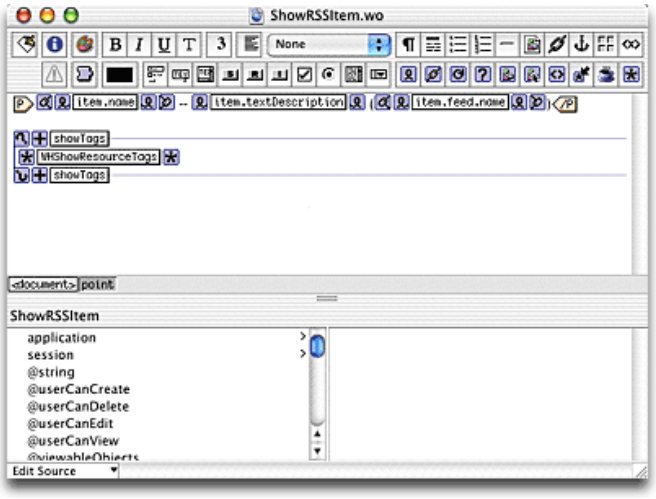
1. Add this method to **RSSItem.java**:

```
public String feedGlobalID() {
    return WHEnterpriseObject.encodedGlobalIDForObject(feed());
}
```

2. In WebObjects Builder, after the description, insert a pair of parentheses, and between them, insert a WOHyperlink. Bind its **actionClass** to **"Display"**, and add a binding called **?resource** set to **item.feedGlobalID**.



3. Inside the WOHyperlink, add a WOString with its **value** set to **item.feed.name**



The final step is to modify the layout dictionary to tell the WHSwitchRenderer components to use ShowRSSItem instead of WHShowResource.

1. Select **layoutDict.plist** in the Resources group in Project Builder.
2. Find this section:

```
renderers = {
    WHChannel = WHShowChannel;
    WHComponentChannel = WHShowComponentChannel;
    WHFetcher = WHShowFetcher;
    WHResource = WHShowResource;
};
```

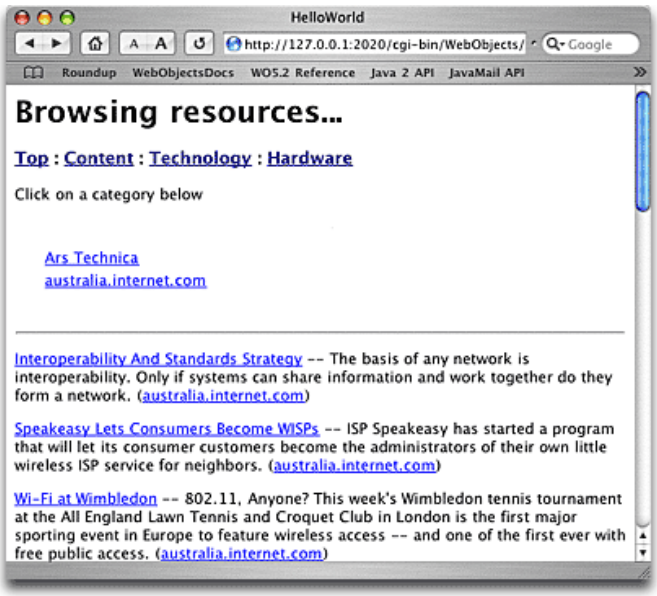
3. Change it so it reads:

```
renderers = {
    WHChannel = WHShowChannel;
    WHComponentChannel = WHShowComponentChannel;
    WHFetcher = WHShowFetcher;
    WHResource = WHShowResource;
    RSSItem = ShowRSSItem;
};
```

4. Build and launch the application, and open this URL in your browser:

```
http://127.0.0.1:2020/cgi-bin/WebObjects/HelloWorld.woa/wa/Drill
```

You'll see your new component is being used to display items.



Note: Since there is a to−many relationship between RSSFeed and RSSItem, you could build a ShowRSSFeed component, and include an option to show the individual items for a feed.

To do this, you would embed a WHSwitchRenderer component inside a WORepetition which iterates over the feed's items, and set the switcher's `object` binding to the item. WireHose will automatically include the proper renderer component to show an item.

At this point the business logic for Hello World is complete. RSS feeds and items are modeled and being imported into the database. The next step will be to customize Hello World's user interface so users can search items, login and create personalized topics for their page.

WireHose layout concepts

All WireHose web components, pages and direct actions are defined in the WireHoseLayoutSupport framework, which also contains application and session−level logic. This framework is only required for WireHose web applications; you can build command−line tools which manipulate resources and tags using just the WireHoseBase framework.

All WireHose application−level logic is accessed via static methods on the WHApplicationHelper class, and all session−level logic is contained in instances of WHSessionHelper.

Note: WireHose is designed so that you can add its frameworks to an existing WebObjects project without having to subclass its application or session classes. WireHose−specific WOApplication and WOSession subclasses are provided, but their use is optional. See the reference documentation for information about adding WireHose support to existing WebObjects classes.

The application helper

WHApplicationHelper is a class that handles application−level WireHose functionality. All its methods are static, so you never instantiate or subclass it. Its primary responsibilities are providing access to system properties, initializing WireHose data structures, providing access to the application's layout dictionary, and handling access control. During startup, the application helper also controls whether and how WireHose will modify EOModels by automatically creating subentities for particular entities.See "Multiple affiliates and auto subentities" for more information about this feature.

Application startup

In response to WOApplication's `ApplicationWillFinishLaunching` notification, the application helper performs a number of initialization tasks. These include getting system properties, setting default values, and logging the values of WireHose−specific properties.

Once it has completed initialization, WHApplicationHelper posts an `ApplicationHelperDidFinishInitializing` notification to declare that it's now safe to access WireHose objects. This is the notification Hello World's Application.java listens for before importing and crawling feeds.

Customizing WHApplicationHelper

WHApplicationHelper provides a delegate interface so you can customize its behavior. It defines several methods which you can use to customize how users are logged in, how guest users get created, and whether or not a user can view, edit or delete objects.

Typically you'll set your application class as WHApplicationHelper's delegate, but any object can be a delegate. You don't need to implement all the methods in the interface or declare that your object implements the interface; WHApplicationHelper will only call the delegate methods you implement.

Customizing authentication

You can override how WireHose associates a user with a session by implementing a delegate method called `userFromRequest`. You can inspect whatever formvalues, cookies or headers you find in the request to determine the user to return. You can also control how guest users are created or fetched through the `guestUserForAffiliate` method.

Customizing access control

The various WireHose layout components, pages and direct actions will query WHApplicationHelper to determine if a user is allowed to view, edit or delete objects. You can implement the `userCanViewObject`, `userCanEditObject` and `userCanDeleteObject` delegate methods to override the default behavior. The `filteredUserViewableObjects` is also available so you can quickly filter entire arrays rather than single objects. Note that these methods will be called often, so they should have as little overhead as possible.

The session helper

Each session in a WireHose application has an associated WHSessionHelper instance, which handles session−specific WireHose properties. It is a "helper" class so you don't have to subclass WHSession to access WireHose features.

The session helper maintains knowledge of the current user and the object currently being edited. It also holds the current WireHose page rendering context, including the current layout and how to render particular items in various areas of assorted pages within the available layouts. WHSessionHelper also handles localization, browser sniffing, login cookie generation and the user's current search string.

Accessing the session helper

The session helper is stored in the session's dictionary. `WHComponent`, `WHSession` and `WHDirectAction` provide access to the session helper via the `helper()` method. You can access the session helper in your own classes by implementing the following method:

```
public WHSessionHelper helper() {
     WHSessionHelper helper = (WHSessionHelper)session().objectForKey(
        WHApplicationHelper.SessionHelperKey);
     if (helper == null) {
         helper = new WHSessionHelper(session());
         session().setObjectForKey(helper,
                WHApplicationHelper.SessionHelperKey);
     }
     return helper;
```

```
}
```

WireHose user interface concepts

Like all WebObjects web applications, a WireHose application consists of pages and components. The WireHoseLayoutSupport framework provides a very dynamic, flexible system for controlling your application's appearance and behavior.

To achieve this flexibility, WireHose introduces the concepts of layouts, pages, wrappers and areas. These are defined in a configuration file known as the application's **layout dictionary**.

Layouts

WireHose provides the ability for your application to have multiple user interface appearances (also known as being "skinnable"). Each separate look in a WireHose application is called a **layout**.

A WireHose application can support multiple branded affiliates from a single codebase, as in an application service provider environment. Or it can allow the user to personalize the look of their page in addition to personalizing its content. You can also use this ability to support multiple output formats, such as XML, HDML, SMIL, RSS, RDF, etc.

You can substitute pages and components for a given layout; the session helper uses the layout dictionary to determine which components to use for each layout.

Pages

Each page is identified by its **canonical name**, which is typically the name of a page's superclass. For example, the canonical name for a search results page is "WHSearchResultsPage". You can provide a subclass of WHSearchResultsPage, such as "MySearchResultsPage", and with an appropriate entry in the layout dictionary, have it be used only when a particular layout is active.

Wrappers

Each layout has an associated **wrapper**, which defines the look for that layout. A wrapper component will include the enclosing `<HTML><BODY>...</BODY></HTML>` tags, and provides a WOComponentContent in which pages are rendered. WireHose page components therefore do not usually contain these tags; rather, they use a WOSwitchComponent to switch in the appropriate wrapper for the current layout via a `helper.currentWrapper` binding. This allows a page component to be used in multiple layouts.

Areas

Wrappers and pages also may define multiple **areas**. For example, a three−column layout may define three areas, "left", "middle" and "right", while another layout may include only a "main" area. Each of a user's channels are mapped to a particular area through its `areaName` property.

The layout dictionary

WHApplicationHelper provides access to the current layout dictionary in its `layoutDictionary` method. The location of the layout dictionary can be specified via the `WHLayoutDict` property, or you can call `setLayoutDictionary` to provide one programmatically in response to the `ApplicationHelperWillFinishInitializing` notification.

Usually you won't access the layout dictionary directly. Rather, the session helper looks up values in the layout dictionary for you to resolve component bindings and values.

What's specified by the layout dictionary

The layout dictionary is used to control which components are switched in for each area, page and layout. It has a very flexible structure: you can define entries which apply to all areas in a layout, all pages in a layout, an area in all layouts, a page in all layouts, an area in a page in a layout, etc.

Components that inherit from `WHComponent` often resolve their bindings through the layout dictionary rather than being set directly by a parent component. A component's `color` binding can resolve to "blue" in one area, and "green" in another, depending on the current area, page and layout.

The current layout

By default, WireHose will use the value of the user's `currentLayout` property to determine which layout to use. You can override this by calling `setCurrentLayout`. WHSessionHelper can also automatically determine which layout to use by sniffing HTTP request headers during its constructor. This is useful for temporarily overriding the user's layout preference depending on the device they are currently using to access the application.

Rapid turnaround

During development, you can disable caching the layout dictionary through the `WHDisableLayoutDictionaryCaching` property. If this property is true, WHApplication will force the layout dictionary to be reloaded before each incoming request is handled. This allows you to make changes in the layout dictionary and see them immediately without re−launching your application.

Using the layout dictionary

WHSessionHelper provides your primary access to the layout dictionary. To resolve bindings such as `helper.@itemInArea.someKey`, the session helper will look in the dictionary at the most specific place possible to resolve the value for `someKey` in the current component. For `itemInArea`, that is the current area, in the current page, in the current layout. If a value isn't found at that location in the dictionary, WHSessionHelper will look at the next most specific place, and so on, until it finds a value. It then caches the value at the original place it looked so the value is immediately available the next time it is needed.

How WireHose resolves layout dictionary values

Here's the order in which WHSessionHelper attempts to resolve values in the layout dictionary for area−level bindings:

1. Area in current page
2. Area in current wrapper
3. Current page
4. Current wrapper
5. Area in default page
6. Area in default wrapper
7. Default page
8. Default wrapper

To resolve page−level bindings, the session helper starts at the current page, and skips the area−level bindings:

1. Current page
2. Current wrapper
3. Default page
4. Default wrapper

Layout dictionary structure

The layout dictionary consists of a nested hierarchy of dictionaries, arrays and constant values:

```
{
    defaults = {
        wrapper = {
            areas = {
                areaName = {
                    componentName = {
                        ...
        pages = {
            pageName = {
                areaName = {
                    componentName = {
                        ...
    };
    layouts = {
        layoutName = {
            wrapper = {
                areas = {
                    areaName = {
                        componentName = {
                            ...
            pages = {
                pageName = {
                    areaName = {
                        componentName = {
                            ...
}
```

See the WHSessionHelper documentation for methods such as `itemInArea(String)` and `itemInPage(String)` for the specific keypaths checked while resolving items.

Editors and renderers

Earlier in this tutorial, you built custom renderers for RSS items and feeds. You added a couple lines to the layout dictionary so WireHose would know to use your renderer instead of the default WHShowResource component. This section describes a few more details about that mechanism.

A common action when rendering a WireHose page is to iterate over a list of objects, and select the appropriate renderer for that object. for example, WHTagDrillerPage iterates over a list of resources which have been tagged with the current tag. For each resource, it uses a `WHSwitchRenderer` to determine which component to use to render it.

Switchers keep the session helper updated as to which object is currently being rendered or edited. The `currentRenderer()` and `currentEditor()` methods look up an area−level dictionary called "renderers" or "editors" to determine which component to use. Here's a sample:

```
pages = {
    MyPage = {
        someArea = {
            renderers = {
                MyPicture = ShowMyPicture;
                Customer = ShowCustomer;
            };
            editors = {
                MyPicture = EditMyPicture;
                Customer = WHBlank; // don't allow editing
            };
```

If it doesn't find an entry for a particular entity, the session helper will look for an entry for each of the current entity's parent entities until it finds a match, so you can have entity−specific renderers as shown in the example.

Note: You can provide your own custom WHSwitchRenderer subclass to enable special behavior or appearance. See the reference documentation for details.

Customizing the user interface

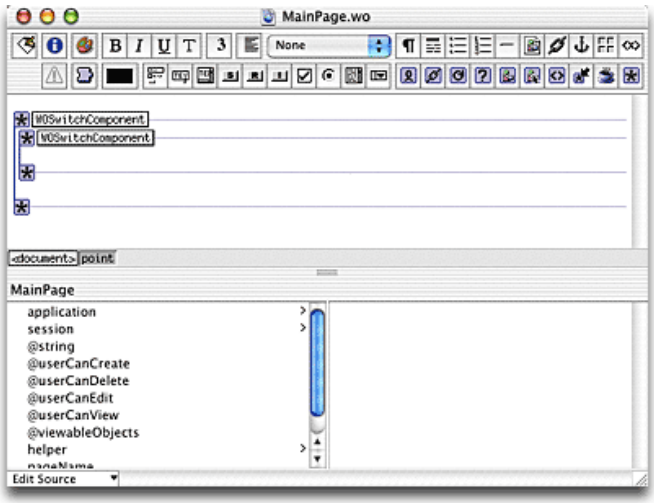
WireHose features a very flexible approach to customizing your application's user interface. The "layout dictionary" defines one or more layouts for an application, and includes entries which determine how component bindings should be resolved for the various areas of each page within a particular layout.

You've already created components to display feeds and items properly, and modified the layout dictionary so your components are used instead of the defaults. In this section, you'll add a search box, and customize how resources can be browsed and searched.

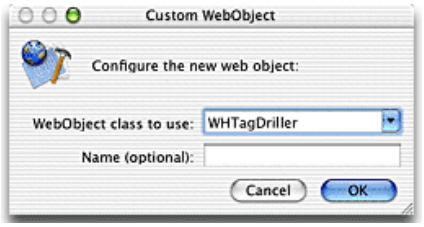
Making the main page

The first customization you'll make to the Hello World user interface will be to allow users to browse available content from the main page. To do this, you'll embed a WHTagDriller component, similar to how a WHTagDrillerPage does. This tagdriller will be limited to showing only items under the "Content" tag.

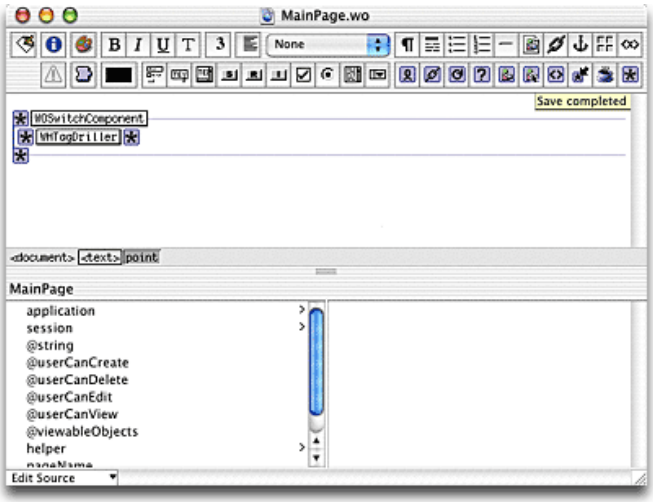
1. Open **MainPage.wo** in WebObjects Builder.



2. Delete the innermost WOSwitchComponent.
3. Choose **Custom WebObject** from the **WebObjects** menu.
4. For "WebObjects class to use:" type **WHTagDriller**, and click **OK**.

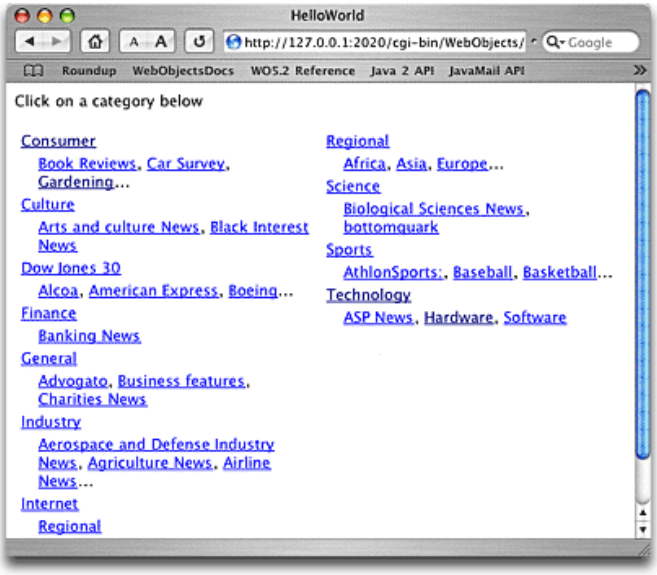


5. Set its **tagPath** to **"Content"** and **showTagPath** to **false**.



6. Launch the Hello World application if necessary. As long as rapid turnaround is enabled, you won't need to rebuild it to see your changes.
7. Open this URL in your browser:

```
http://127.0.0.1:2020/
```



Your browser will be redirected to the MyHomePage direct action, which displays

the current WHMainPage component. This entry in the layout dictionary causes your MainPage component to be used instead of the default WHMainPage:

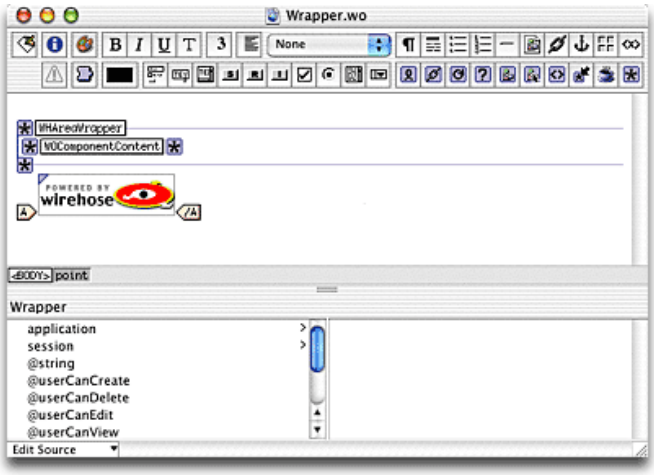
```
defaults = {
   ...
    wrapper = {
       ...
    };
    pages = {
       ...
        WHMainPage = {
            pageName = MainPage;
```

Adding keyword searching

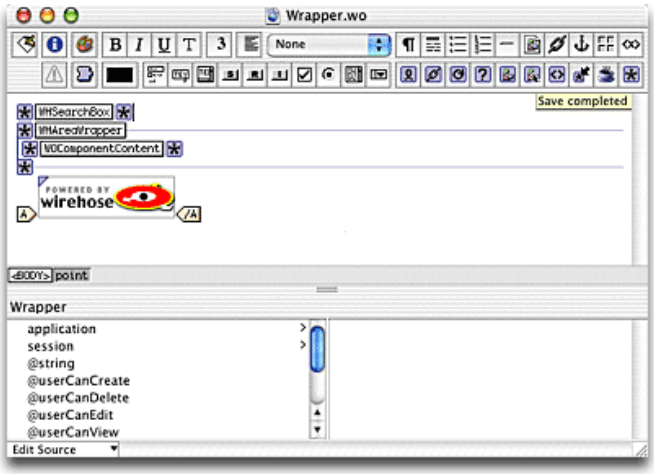
Next, you'll add a search box so users can search items and feeds by keywords. The search box will be added to the Wrapper component so it will be available on all pages by default.

Adding the search box

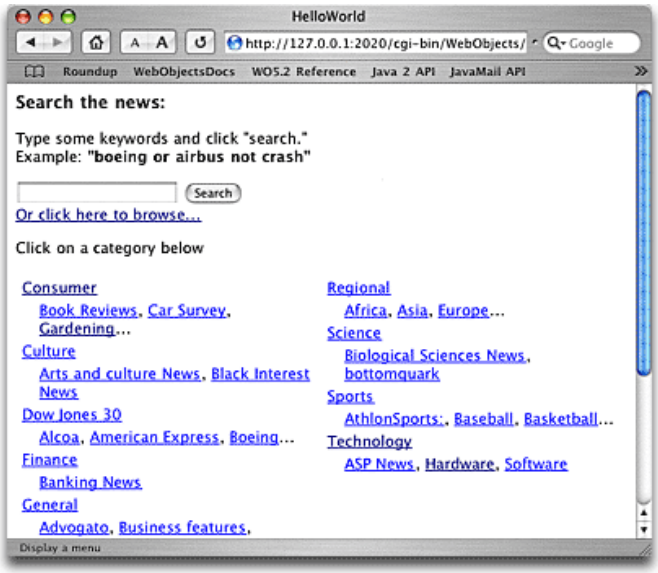
1. Open **Wrapper.wo** in WebObjects Builder.



2. Choose **Custom WebObject** from the **WebObjects** menu.
3. For "WebObjects class to use:" type **WHSearchBox**, and click **OK**.



4. Reload the page in your browser.



Customizing the search prompt

The default prompt for WHSearchBox isn't suitable for Hello World, so next you'll change it.

WireHose has extensive localization support which allows you to control at a very fine level how strings are localized for particular components, languages, pages and areas. You can also define non−localized strings in the layout dictionary, which is the approach we'll take here.

> **Note:** See the WHSessionHelper reference documentation for details about WireHose localization support.

WHSearchBox defines three strings, called "search", "prompt" and "orClickToBrowse".

You'll override just the prompt string.

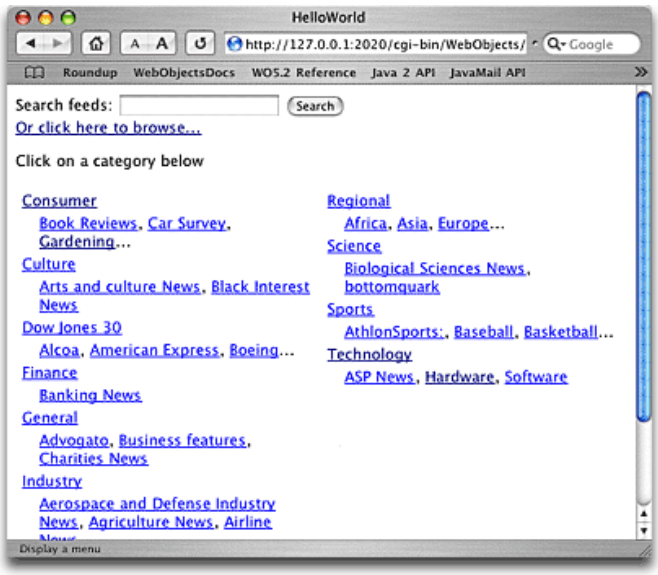
1. Find this entry in the layout dictionary:

```
WHSearchBox = {
    showBrowsePrompt = YES;
};
```

   Change it so it reads:

```
WHSearchBox = {
    showBrowsePrompt = YES;
    strings = {
        prompt = "Search feeds: ";
    };
};
```

2. Relaunch Hello World and reload the page in your browser.



Customizing the search box on specific pages

Since the main page and the tag driller page in Hello World already allow users to browse through available resources, the "Or click here to browse..." link is redundant on those pages. WHSearchBox lets you control whether this link is shown via its `showBrowsePrompt` binding.

Ordinarily you'd set the showBrowsePrompt to false if you were to embed the search box directly into the main page and tag driller page. But in this case, the search box is embedded into the wrapper, and Hello World doesn't even have its own tag driller page component anyway.

Another technique might be to add a method to Wrapper.java which returns true or false depending on the current page. This method might look like this:

```
public boolean showBrowsePrompt() {
    return (context().page() instanceof MainPage ||
        context().page() instanceof WHTagDrillerPage);
}
```

or this:

```
public boolean showBrowsePrompt() {
    return ("WHMainPage".equals(helper().currentPage()) ||
        "WHTagDrillerPage".equals(helper().currentPage());
}
```

However, this approach can be cumbersome to maintain. WireHose provides an alternative approach, which has the advantage of not requiring custom code: resolving the binding through the layout dictionary.

This entry in the layout dictionary sets WHSearchBox's showBrowsePrompt to true in all areas on all pages in all layouts by default:

```
defaults = {
   ...
    WHSearchBox = {
        showBrowsePrompt = YES;
```

```
...
```

You'll modify the layout dictionary so showBrowsePrompt resolves to false on the main page and the tag driller page.

1. First, set it for the main page. Find this entry in the layout dictionary:

```
WHMainPage = {
    pageName = MainPage;
   ...
```

Change it so it reads:

```
WHMainPage = {
    pageName = MainPage;
    WHSearchBox = {
        showBrowsePrompt = NO;
    };
   ...
```

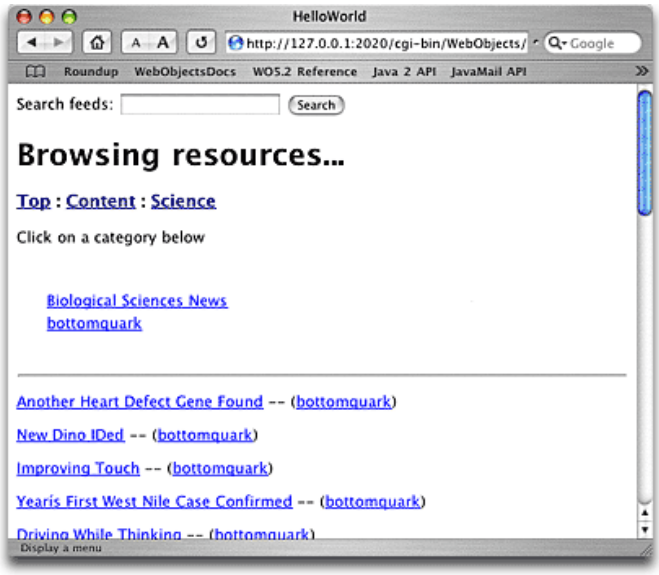
2. Next, set it on the tag driller page. Find this entry in the layout dictionary:

```
WHTagDrillerPage = {
    pageName = WHTagDrillerPage;
   ...
```

Change it so it reads:

```
WHTagDrillerPage = {
    pageName = WHTagDrillerPage;
    WHSearchBox = {
        showBrowsePrompt = NO;
    };
   ...
```

3. Reload the page in your browser.



How this works: WHSearchBox includes a WOConditional to determine whether to show the browse prompt. The conditional's **condition** is bound to **showBrowsePrompt**. WHSearchBox.java defines this method:

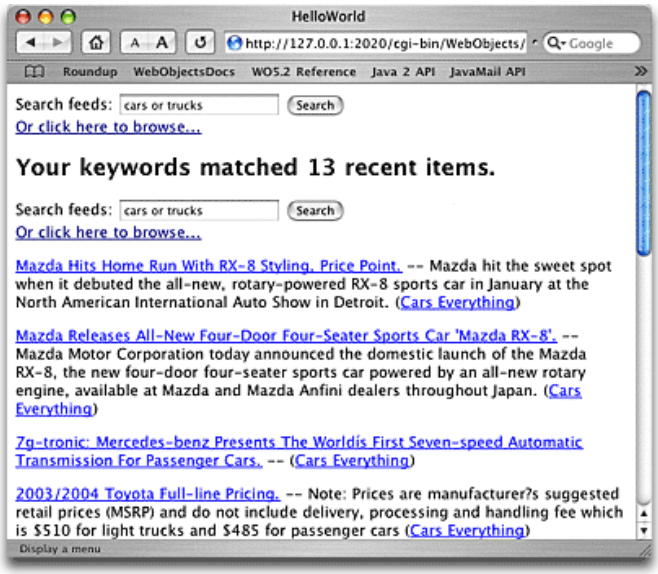
```
public boolean showBrowsePrompt() {
    return booleanForBinding("showBrowsePrompt");
}
```

If WHSearchBox's showBrowsePrompt binding is bound directly to true or false, WHComponent's `booleanForBinding` method will return that value. But if the binding is left unbound, `booleanForBinding` will resolve the value via the layout dictionary.

You can use this technique in your own components. And if you're resolving simple string bindings, you don't even have to implement a method in your code −− WHComponent will resolve the value via the layout dictionary automatically. See the reference documentation for WHComponent for details.

Removing the search box from a specific page

The WHSearchResultsPage component includes a search box by default.



This interferes with Hello World's user interface because a search box is already included within the wrapper. In this step you'll remove the search box through another entry in the layout dictionary.

1. Find this entry in the layout dictionary:

```
WHSearchResultsPage = {
    pageName = WHSearchResultsPage;
    showSearchBox = YES;
};
```

   Change it so it reads:

```
WHSearchResultsPage = {
    pageName = WHSearchResultsPage;
    showSearchBox = NO;
};
```

2. Reload the page in your browser.

How this works: Just like WHSearchBox, the WHSearchResultsPage includes a WOConditional to determine whether to include a search box. The conditional's **condition** is bound to **showSearchBox**. Just like WHSearchBox.java, WHSearchResultsPage.java defines this method:

```
public boolean showSearchBox() {
    return booleanForBinding("showSearchBox");
}
```

However, since page−level components are never embedded in another component, you can't set any bindings directly. Resolving page−level bindings through the layout dictionary lets you customize page components without writing any code.

Adding personalization

In this section, you'll build a custom tag driller page, which allows guest users to click on an "add this to my page" when they find a category they like, and receive a signup page. This page will validate the user's login and password, and create a new user in the database. Once the new user has been inserted into the database, you'll create a new custom fetcher which will display items from the category the user selected.

Add this to my page

WHTagDrillerPage determines whether or not to show an "add this to my page" button by calling WHApplicationHelper's `userCanEditObject` method, with the user as the object in question. By default, the guest user isn't permitted to edit anything, so you'll override that behavior by implementing a method from the WHApplicationHelper.Delegate interface.

1. Uncomment this line in **Application.java**'s constructor:

   ```
   WHApplicationHelper.setDelegate(this);
   ```
2. Add this method:

   ```
   public boolean userCanEditObject(WHUser user, Object object, WOContext context) {
       if (user.isGuest() && user.equals(object) &&
           "WHTagDrillerPage".equals(context.page().valueForKey("pageName"))) {
           return true;
       } else {
           return super.userCanEditObject(user, object, context);
       }
   }
   ```

   When the tag driller page checks to see if the user can edit itself, WHApplicationHelper will call this method, and it will return true, so the button will be displayed.
3. Build and launch the application, then open this URL in your browser:

   ```
   http://127.0.0.1:2020/
   ```
4. When you browse to a tag which has matching resources, the "Add this to my page"

will appear.

In the next section, you'll build a custom subclass of WHTagDrillerPage so Hello World can create a new user when the button is clicked.

Building TagDrillerPage

The "add this to my page" button is bound to WHTagDrillerPage's `addTag` method. In this step, you'll subclass WHTagDrillerPage and override the addTag method so that it returns a signup page if the current user is a guest.

Because the layout dictionary may specify that pages may be substituted in a particular layout, here we'll use the session helper's `nameForPage` method to determine the actual page class to return from `addTag()`.

1. Select the **Web Components** group in the Files pane.
2. Choose **New File...** from the **File** menu. Scroll down to the WireHose **Page** template, and click **Next**.
3. Name it **TagDrillerPage**, add it to the Application Server target in the Hello World target, and click **Finish**.
4. Edit **TagDrillerPage.java** so it reads like this:

```
public class TagDrillerPage extends WHTagDrillerPage {

    public TagDrillerPage(WOContext context) {
```

```
        super(context);
    }

    public boolean shouldUseAlternateTemplate() {
        return true;
    }

    public WOActionResults addTag() {
        if (helper().user().isGuest()) {
            SignupPage nextPage =
              (SignupPage)pageWithName(helper().nameForPage("SignupPage"));
            nextPage.setTag(tag());
            return nextPage;
        } else {
            return super.addTag();
        }
    }
}
```

5. Find this entry in the layout dictionary:

```
WHTagDrillerPage = {
    ...
    pageName = WHTagDrillerPage;
    ...
```

Change it so it reads:

```
WHTagDrillerPage = {
    ...
    pageName = TagDrillerPage;
    ...
```

WireHose will now use TagDrillerPage instead of WHTagDrillerPage.

How this works: In this example, we're subclassing WHTagDrillerPage to modify behavior, but not duplicating any of its HTML or .wod definitions. The page will still look just like a WHTagDrillerPage, though.

The key is the `shouldUseAlternateTemplate` method. Since it returns true here, TagDrillerPage will ignore its normal HTML and .wod definitions, and instead get them from two methods called

`templateHTMLString` and `templateDefinitionString`. By default, these methods will return the HTML and .wod from the component's superclass. So TagDrillerPage ends up using WHTagDrillerPage's template.

You can override `templateHTMLString` and `templateDefinitionString` to return strings from any source. A very flexible and powerful technique is to store HTML and component definitions in a database.

Building SignupPage

The SignupPage component takes the current tag from the TagDrillerPage, then asks the user to enter a login and password, and re−enter the password. The prompts and messages will be localizable instead of hard−coded into the page's HTML. In addition, the fetchMatchingUsers method (which checks to see if a user has typed a login which is already in the database) and the createUser method both dynamically resolve the current user entity.

Adding the component

1. Add this entry to the layout dictionary in the pages section:

```
pages = {
    SignupPage = {
        pageName = SignupPage;
    };
    WHMainPage = {
    ....
```

2. Select the **Web Components** group in the Files pane.
3. Choose **New File...** from the **File** menu. Scroll down to the WireHose **Page** template, and click **Next**.
4. Name it **SignupPage**, add it to the Application Server target in the Hello World target, and click **Finish**.
5. Choose **New File...** from the **File** menu. Scroll down to the WireHose **Strings File** template, and click **Next**.
6. Name it **SignupPage**, add it to the Application Server target in the Hello World target, and click **Finish**.

Building the UI

SignupPage is a typical WebObjects component, with the exception that the message prompt and field and button labels will be localizable rather than hard−coded into the HTML or Java code. You can resolve localized strings with a simple **@string.*key*** or **self.@string.*key***.

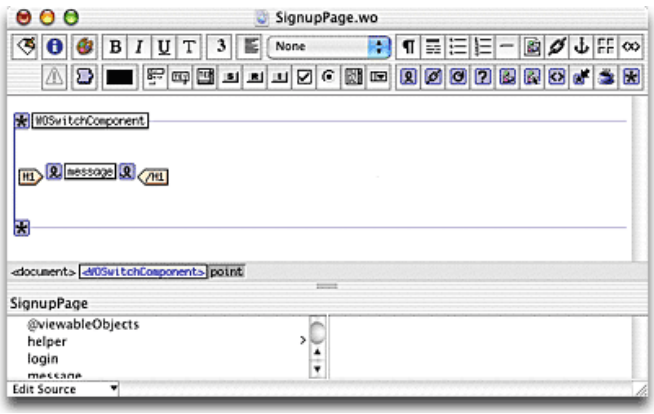
Note: Other useful bindings provided by WHComponent include **@userCanEdit.*keyPath*** and **@userCanView.*keyPath***, which return true or false by querying the application helper (and its delegate, if set) as to whether or not the object identified by ***keyPath*** in the current context can be edited, viewed or deleted by the current user. The ***keyPath*** is any keypath which is currently valid, so you can bind things like @userCanEdit.helper.editingObject to a WOConditional.

Note: WebObjects Builder on Windows will not allow bindings which start with an "@", such as **@string.*key*,** so WireHose also supports **self.@string.*key***. There are no performance penalties for using WHComponent's **self.@string.*key*** or **self.@userCanView.*keyPath*** bindings over **@string.*key*** or **@userCanView.*keyPath***.

1. Add this line to **SignupPage.java**:

   ```
   public String message;
   ```

2. Open **SignupPage.wo** in WebObjects Builder.
3. Select the inner WOSwitchComponent and delete it.
4. Inside the remaining WOSwitchComponent, add an H1 heading.
5. Inside the heading, add a WOString and set its **value** to **message**. This will be the prompt for the signup page.



6. Change SignupPage's constructor so it reads like this:

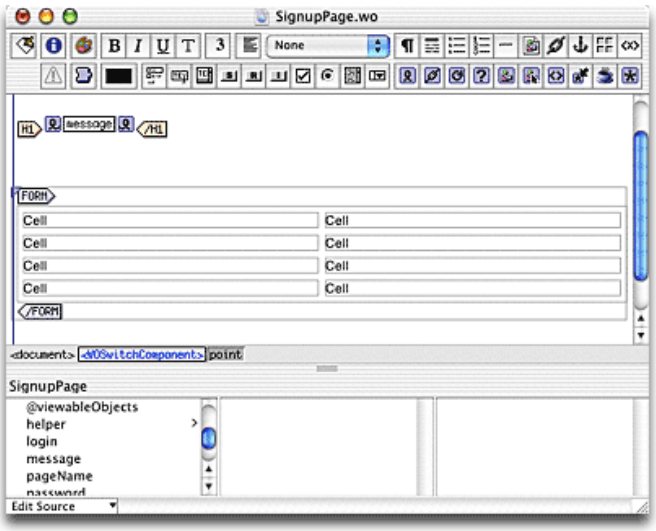
   ```
   public SignupPage(WOContext context) {
   ```

```
        super(context);
        message = helper().stringInComponent(this, "signup");
    }
```

7. Add this line to **SignupPage.strings**:

```
signup = "Sign up for a new account";
```

8. Underneath the heading in **SignupPage.wo**, add a WOForm. Set its **action** to **createAccount**, and set **multipleSubmit** to **true**.
9. Delete the text inside the form and add a new table. Make it 4 rows, 2 columns, 0 border, 6 spacing, 0 padding. Uncheck both "First row cells are header cells (<TH>)" and "Second row is wrapped in a WORepetition". Click **OK**.



10. Add three WOString components to the left column in the first three rows of the table. Set the **value** for the first to **@string.login**. Set the value for the second to **@string.password** and the third to **@string.passwordAgain**
11. Add these lines to **SignupPage.strings**:

```
login = "make up a user name";
password = "make up a password";
passwordAgain = "type password again";
signupButton = "Sign Up";
wantCookie = "save my login and password in a cookie";
```

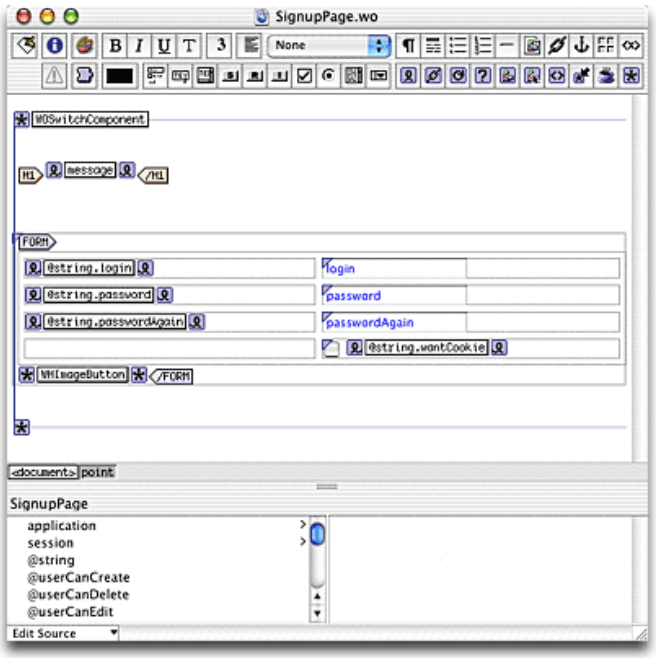
12. Add these lines to **SignupPage.java**:

```
public String login, password, passwordAgain;
public boolean shouldSaveCookie = false;
```

13. Select the right column of the first row of the table, and add a WOTextField and set

its **value** to **login**

14. In the second row, add a WOTextField and set its **value** to **password**. Choose the Static Inspector on the Inspector pallete, and click **Password Field (invisible typing)**
15. In the third row, add a WOTextField and set its **value** to **passwordAgain**. Make this a password field also.
16. In the last row, add a WOCheckBox and set its **checked** to **shouldSaveCookie**. Add a WOString and set its **value** to **@string.wantCookie**
17. Select inside the form, after the table, and choose **Custom WebObject** from the **WebObjects** menu. For "WebObjects class to use:" type **WHImageButton**, and click **OK**.
18. Set the image button's **action** to **createAccount**, set its **filename** to **helper.@itemInPage.signupButton**, set its **framework** to **helper.@itemInPage.framework** and set its **label** to **@string.signupButton**



Note: Since WireHose components can be used in any number of layouts, which may have special graphical needs for buttons or links, WireHose includes the WHImageButton component. WHImageButton will render itself as a hypertext link, a linked image, a graphical submit button, or a plain submit button, depending on whether it's currently in a form or not, and if its **filename** binding resolves to a string or not. Bindings such as **filename** and **framework** are typically resolved through the layout dictionary rather than being bound directly.

Writing the code

Next, you'll add the code that makes the signup page work.

1. First, add this code to **SignupPage.java** so the signup page knows which tag was selected in the tag driller page:

```
private WHTag _tag;

public void setTag(WHTag value) {
    _tag = value;
}

public WHTag tag() {
    return _tag;
}
```

2. SignupPage will need to sanity check user input. The `createAccount` method will use an `errorMsg` method to return error messages to the user:

```
public WOActionResults errorMsg(String msg) {
    message = helper().stringInComponent(this, msg);
    password = "";
    passwordAgain = "";
    return context().page();
}
```

3. Add these lines to **SignupPage.strings**:

```
missingField = "Please enter all fields";
passwordMismatch = "Your password did not match";
reservedID = "Sorry, that login is reserved";
```

4. Part of the sanity checking is to verify that the user hasn't entered a login that was already used in the database. Add this method to **SignupPage.java**:

```
private NSArray fetchMatchingUsers() {
    EOQualifier q = new EOKeyValueQualifier(
      "login", EOQualifier.QualifierOperatorEqual, login);
    EOFetchSpecification fs =
      new EOFetchSpecification(WHApplicationHelper.userEntityName(), q, null, true, t
    return session().defaultEditingContext().objectsWithFetchSpecification(fs);
}
```

And finally, the `createAccount` method itself:

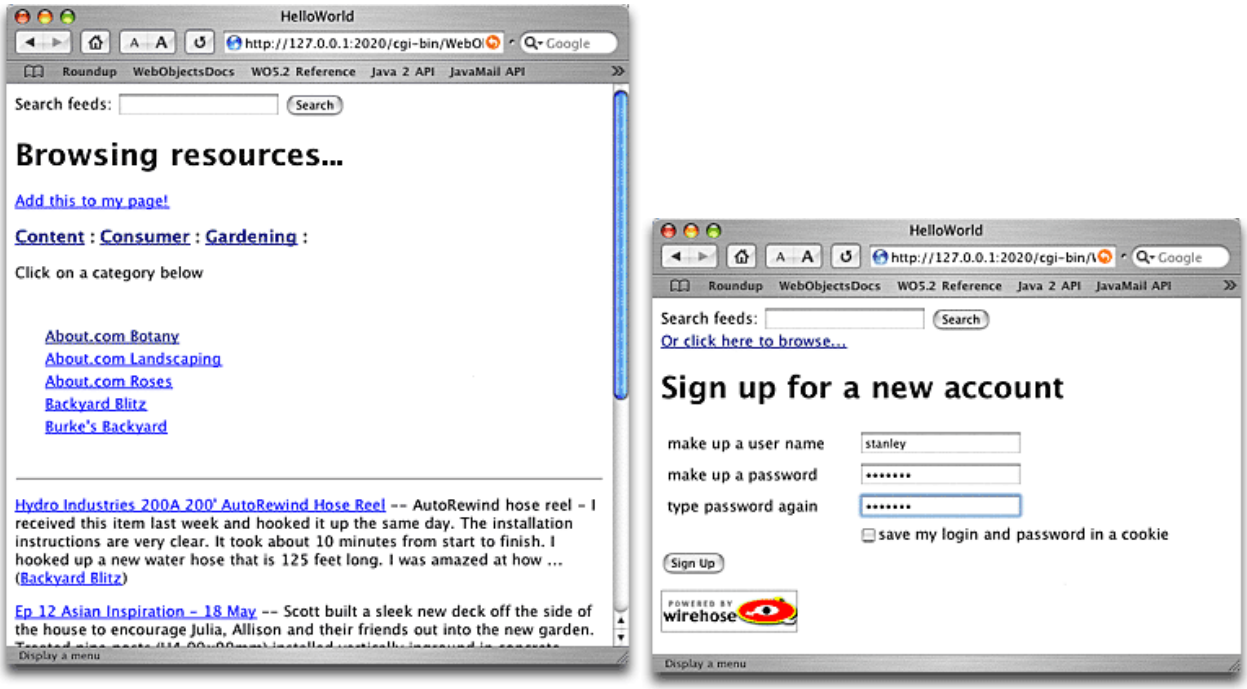
```
public WOActionResults createAccount() {

    // sanity check user input
    if (login == null || password == null || passwordAgain == null ||
         "".equals(login) || "".equals(password) || "".equals(passwordAgain)) {
        return errorMsg("missingField");
    }

    // make sure passwords match
    if (!password.equals(passwordAgain)) {
        return errorMsg("passwordMismatch");
    }

    // make sure login wasn't already used
    NSArray users = fetchMatchingUsers();
    if (users.count() != 0) {
        return errorMsg("reservedID");
    } else {

        // never know what the current user entity might be
        WHUser user = (WHUser)WHEnterpriseObject.createAndInsertInstance(
            session().defaultEditingContext(),
            WHApplicationHelper.userEntityName(),
            WHApplicationHelper.defaultAffiliate());

        user.setDateLastLogin(new NSTimestamp());
        user.setLogin(login);
        user.setPassword(password);
        session().defaultEditingContext().saveChanges();

        // replace the guest user for this session
        helper().setUser(user);

        WHTagDrillerPage nextPage =
            (WHTagDrillerPage)pageWithName(helper().nameForPage("WHTagDrillerPage"));
        nextPage.setTag(tag());
        WOActionResults response = nextPage.addTag();

        // set a login cookie if the user asked for it
        if (shouldSaveCookie) {
            return helper().addLoginCookieToResponse(response.generateResponse());
```

5.
```
        } else {
            return response;
        }
    }
}
```

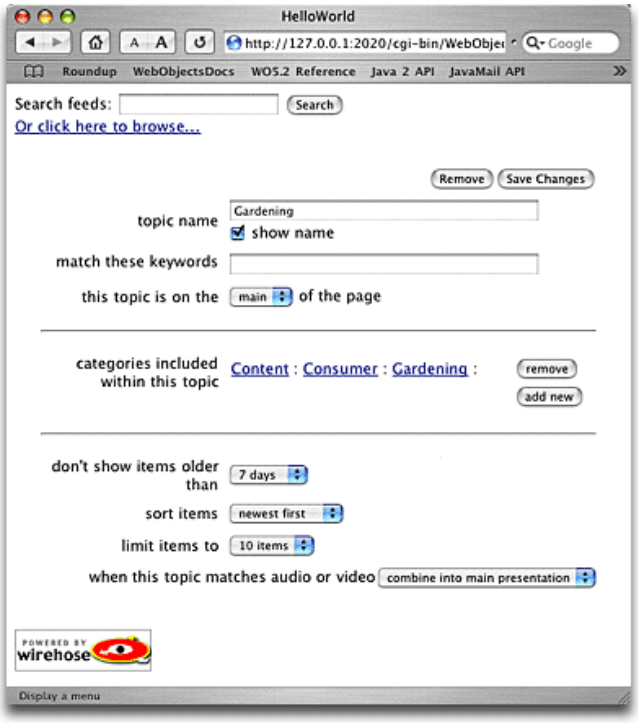
6. Build the application, launch it, and open this URL in your browser:

```
http://127.0.0.1:2020/
```

7. Browse through the content, and when you get to an interesting category, click the "Add this to my page" button and create a new account.



8. After creating your account, you'll be returned to the WHEditObjectPage, where you can adjust settings on the newly−created channel.



Note: Several WireHose components, including the WHEditFetcher shown here, have a binding called **hideDetails**. If this binding resolves to true, then WHEditFetcher will show an abbreviated version of itself. See the **NewsDemo** sample application for an example which allows the user to control this with a pair of hide details/show details controls.

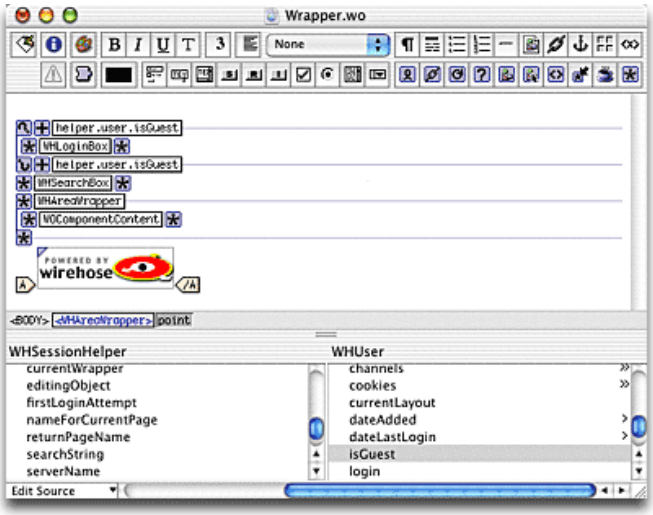
Finishing the user interface

The final step in building Hello World will be to clean up the user interface and add some navigation. You'll add a login panel so users with accounts can login, and customize the view for logged in users.

Adding a login panel

Now we'll add a login panel to Hello World. The panel will be added to the wrapper so it will show up on every page.

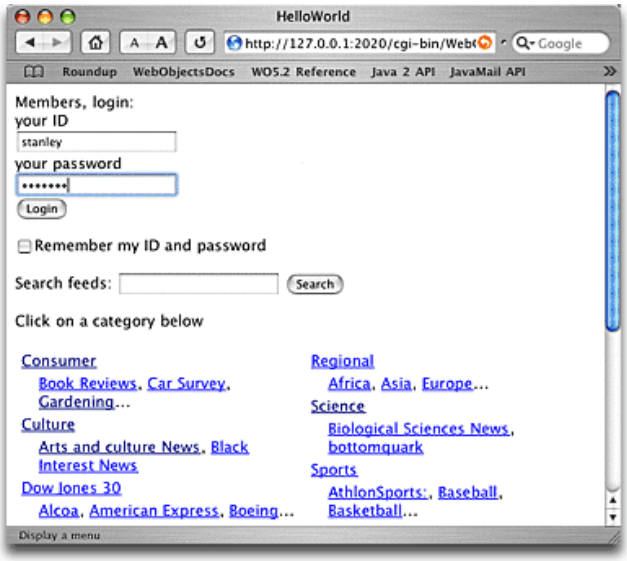
1. Open **Wrapper.wo** in WebObjects Builder.
2. Add a WOConditional, and set its **value** to **helper.user.isGuest**
3. Select inside the conditional, and choose **Custom WebObject** from the **WebObjects** menu. For "WebObjects class to use:" type **WHLoginBox**, and click **OK**.



4. If you haven't quit and relaunched Hello World, open this link in your browser to logout the current user:

   ```
   http://127.0.0.1:2020/cgi-bin/WebObjects/HelloWorld.woa/wa/Logout
   ```

5. Now you can use the new login box to sign in to the account you created earlier.



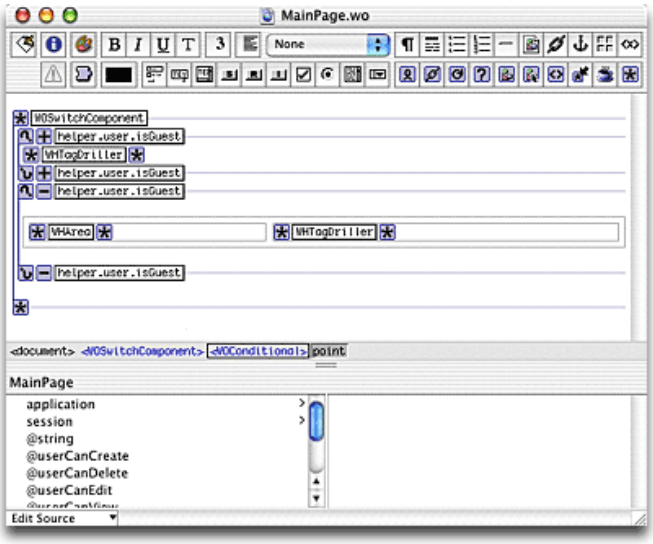
Customizing the main page

We'll create a different appearance for the main page depending on whether the user is a guest or not. The page guest users see will be dominated by a Yahoo−style collection of categories, while the page registered users see will be dominated by their selected topics of interest.

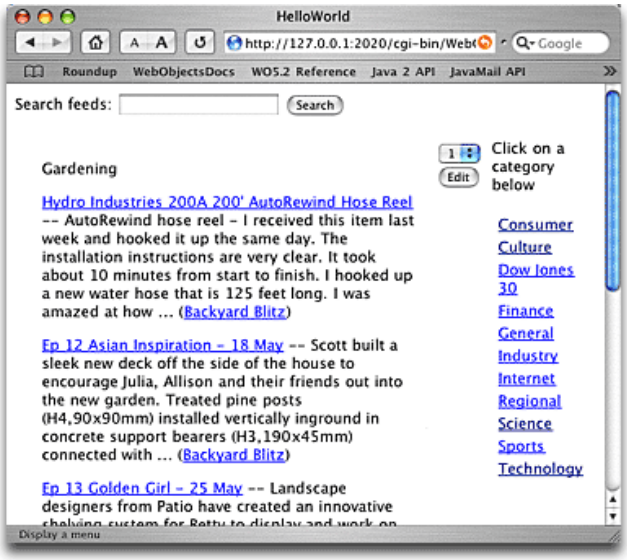
1. Open **MainPage.wo** in WebObjects Builder.
2. Select the WHTagDriller component, and add a WOConditional around it. Set its **condition** to **helper.user.isGuest**
3. Add another WOConditional. Set its **condition** to **helper.user.isGuest** and set its **negate** to **true**
4. Inside the conditional, add a new table with 1 row, 2 columns, 0 border 6 spacing and 0 padding. Uncheck both "First row cells are header cells (<TH>)" and "Second row is wrapped in a WORepetition" and click **OK**.
5. Select the left cell, and choose **Custom WebObject** from the **WebObjects** menu. For "WebObjects class to use:" type **WHArea**, and click **OK**. Set its **areaName** to **"main"**

   > **Note:** The WHArea component iterates over the user's channels which have been mapped to its **areaName** value, and renders each with a WHSwitchRenderer. Hello World's layout has just one area, named "main".

6. Select the right cell, and choose **Custom WebObject** from the **WebObjects** menu. For "WebObjects class to use:" type **WHTagDriller**, and click **OK**.
7. Set its **maxChildTags** to **0**, **numCols** to **1** and its **tagPath** to **"Content"**



8. If you haven't logged in yet, login, or just reload the page in your browser.

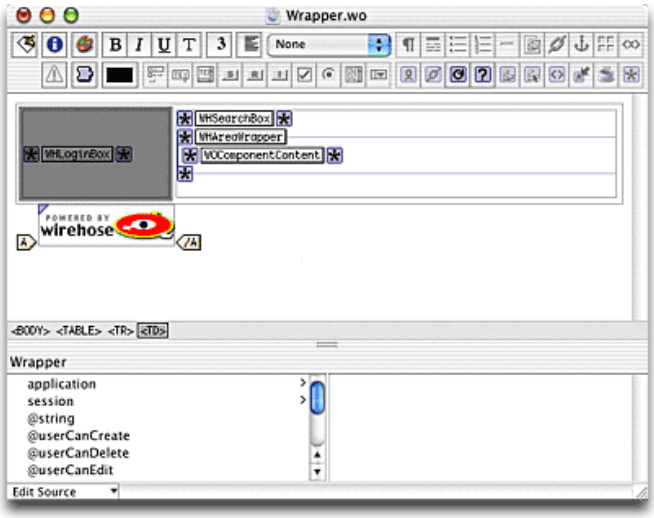


Adding navigation

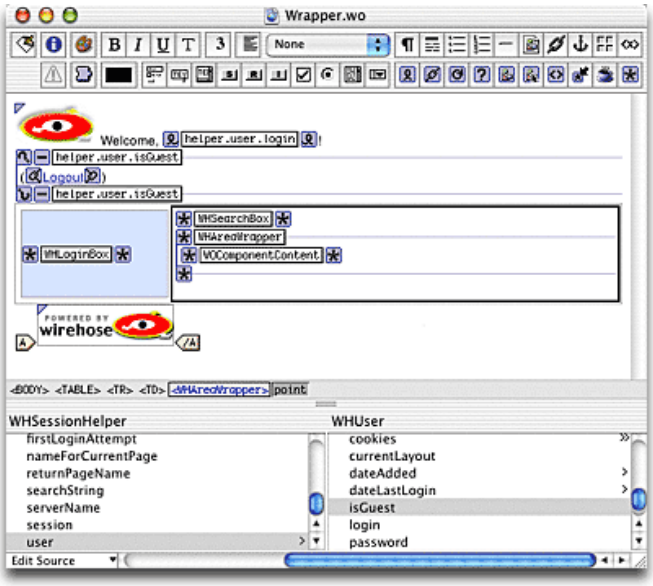
As a final step in the development of Hello World, we'll clean up the wrapper a bit and add some navigation.

1. Open **Wrapper.wo** in WebObjects Builder
2. Add a table, with 1 row and 2 columns.
3. Select the WHLoginBox, and cut and paste it into the left cell of the table. Set the width of the cell to 25 percent.
4. Remove the WOConditional which used to wrap the login box.

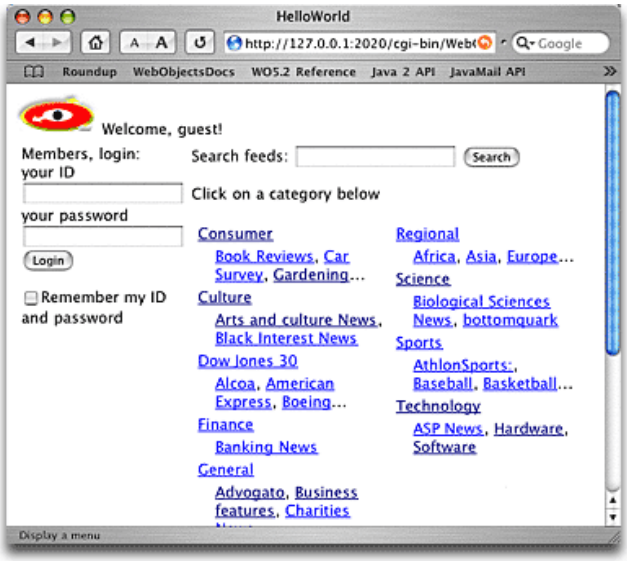
5. Cut and paste the WHSearchBox and WHAreaWrapper into the right cell. Set the cell's width to 75 percent.
6. Select the entire left cell by clicking on the <TD> in the path view.

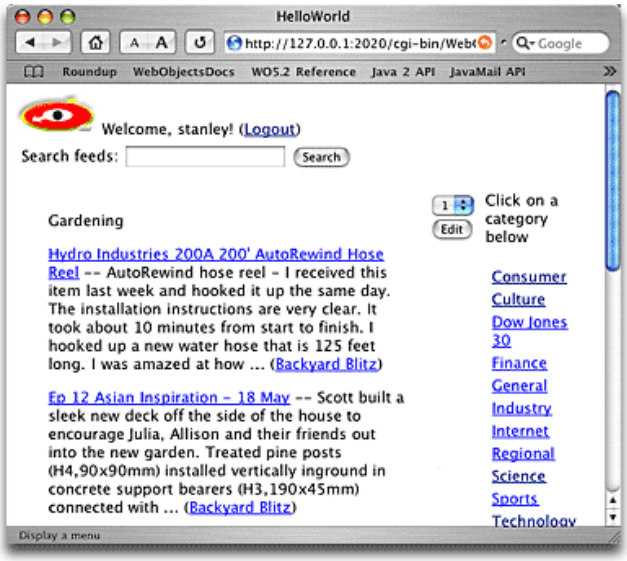


7. Add a WOConditional to wrap the entire cell (not just its contents), and set its **condition** to **helper.user.isGuest**
8. At the top of the page, add a WOActiveImage. Set its **actionClass** to **"MyHomePage"**, set its **framework** to **"WireHoseLayoutSupport"**, and its **filename** to **"wirehose_small_white.gif"**. Add a binding named **border** and set it to **0**.
9. To the right of the image, type **Welcome,** add a WOString with its value set to **helper.user.login** and type an exclamation point and **(Logout)**.
10. Wrap the **(Logout)** text in a WOConditional. Set its **condition** to **helper.user.isGuest** and its **negate** to **true**.
11. Select the word **Logout** and add a WOHyperlink with its **actionClass** set to **"Logout"**.



12. Reload the page and explore Hello World.




HelloWorld
http://127.0.0.1:2020/cgi-bin/Web
Google
Roundup
WebObjectsDocs
WO5.2 Reference
Java 2 API
JavaMail API
Welcome, stanley! (Logout)
Search feeds:
Search
Gardening
1
Edit
Click on a category below
Hydro Industries 200A 200' AutoRewind Hose Reel -- AutoRewind hose reel - I received this item last week and hooked it up the same day. The installation instructions are very clear. It took about 10 minutes from start to finish. I hooked up a new water hose that is 125 feet long. I was amazed at how ... (Backyard Blitz)
Ep 12 Asian Inspiration - 18 May -- Scott built a sleek new deck off the side of the house to encourage Julia, Allison and their friends out into the new garden. Treated pine posts (H4,90x90mm) installed vertically inground in concrete support bearers (H3,190x45mm) connected with ... (Backyard Blitz)
Consumer
Culture
Dow Jones 30
Finance
General
Industry
Internet
Regional
Science
Sports
Technology
Display a menu

Further exploration

You've now completed building the Hello World application. In the process, you've learned about important WireHose business logic concepts such as taggable and indexable resources, channels, fetchers and tags. You've modeled WireHose resources and built custom renderers to show them properly.

You also put into practice key concepts that give WireHose applications their user interface flexibility, such as the session helper, the layout dictionary, pages, wrappers and localization.

There's much more to WireHose than what's been covered so far. This section will describe some of the other features available in WireHose that are beyond the scope of this tutorial

Component channels

WHComponentChannel allows you to use any WOComponent as a channel.

Qualifier fetchers

WHQualifierFetcher instances are channels that fetch enterprise objects based on an arbitrary qualifier. Instead of fetching resources based on their tags or keywords, a qualifier fetcher uses a string such as "lastName = 'Weber' AND supervisor.firstName = 'Stanley'". Subclasses of WHQualifierFetcher can provide sort orderings and qualifier bindings (enabling queries such as "lastName = %@").

Qualifier fetchers are especially good for creating enterprise portals since you can easily create a collection of channels that give users access to widely varying enterprise data such as recent sales, inventory, current shipments, expiring contracts, etc., without having to write any WireHose−specific business logic.

Streaming resources

Resources in WireHose are generally divided into two groups: streaming resources, which are those which have a duration when rendered, such as video or audio clips, and non−streaming resources, which are everything else.

WHTagFetcher has special support through its groupStreaming property for determining how to render streaming resources. There are three different options available:

- Allow the user to view individual streaming resources matched by this fetcher.
- Present all streaming resources matched by this fetcher in a single presentation.
- Combine the streaming resources from this fetcher, and all others with the same setting, into a single presentation for the user.

Revision tracking

WHRevision is a subclass of WHTag which provides support for versioning WHTaggable objects. A WHRevision tag represents a particular resource, and can be assigned to another resource to indicate that a resource is a revision of another. Since resources can have multiple tags, a given resource can be a revision of multiple other resources, and a given resource can have multiple revisions (and revisions of revisions, and so on).

WHRevision provides three key methods for handling versioning: `makeRevision` makes one resource a revision of another; `revisionsForResource` returns all revisions of a particular resource; and `originalsForRevision` returns the array of objects a particular revision is a revision of.

Access control

The WireHoseEngageSupport framework is an optional collection of classes which work together to provide roles−based access control for taggable objects.

WHEngageTag is a WHTag subclass which implements WHTaggable and WHIndexable to implement access control.

WHEngageUser is a subclass of WHUser. Users can belong to multiple groups, and also have their own group so you can grant permissions directly to an individual user.

WHGroup is a tag whcih represents groups of users which have permission to perform operations on taggable objects. Groups can be arranged in any hierarchy required; membership in a group implies membership in all of its ancestor groups. Everyone is a member of the "Public" group.

WHPermission is a tag which is assigned to taggable objects to indicate that members of a specific group can perform a specific operation on the tagged objects. Permissions are inheritable by default; if an inheritable permission is assigned to a tag, then all resources tagged with that tag, or any of its descendent tags, will share the same permission as if it had been assigned to each resource directly.

WHOperation objects represent a particular operation which can be exercised on a taggable object, such as viewing, editing or deleting. Operations are hierarchical; the "Manipulate" operation implies its descendents, "View", "Edit," "Delete" and "Assign". The "Assign" operation models assigning tags to taggable objects.

Tag templates

WHEngageTag provides methods to use templates to create a collection of tags based on the template at an arbitrary place in the tag hierarchy. In its most basic form, creating tags from a template just duplicates the template's descendent tags.

Templates can be built from ordinary tags, but they achieve much of their power when used in conjunction with WHGroupTemplate and WHPermissionTemplate tags.

A group template can have permissions associated with it to control who can view or edit the template. The "assign" permission controls who can instantiate tags using the template.

In addition to having permissions assigned to a group template, you can also assign permission templates. Permission templates are used to create permissions when the group template is instantiated.

Group templates can have descendent group templates, which become descendent groups when instantiated. You can assign permission templates to a descendent group template and have the permissions apply to the descendent group when the parent template is instantiated.

Bookmarkable URLs

WireHose has several featurs which enable your applications to have clean, bookmarkable URLs such as "/WireHoseDemo/MyHomePage" and still provide personalized sessions.

Cookies

WireHose applications don't store sessionIDs in the URL unless the user has disabled cookies in the browser. WireHose provides a mechanism for for automatically detecting whether cookies are enabled in a client's browser, and controlling whether session IDs are visible in URLs accordingly.

When called for the first time during a session, a WireHose direct action will check to see if cookies should be enabled. If necessary, the browser will be redirected to the GotCookies direct action, providing a session id in both the URL and via cookies. During the GotCookies direct action, the session's storesIDsInCookies and storesIDsInURLs are set appropriately, and another redirect is issued back to the original direct action.

Rewrite rules

WHHyperlink is a replacement for WOHyperlink which includes support for URL−rewriting similar to Apache's mod_rewrite. This rewriting is applied to URLs generated by your application rather than incoming browser requests as with mod_rewrite.

You can provide a perl−style regular expression that gets applied to the URL, providing an outgoing−URL counterpart to Apache's mod_rewrite (which is applied to incoming request URLs).

For example, this rule

```
s'cgi−bin/WebObjects/WireHoseTest.woa/wa/Drill(?:\\?path=|)'Resources/'
```

will transform a URL like

```
http://www.wirehose.com/cgi−bin/WebObjects/WireHoseTest.woa/wa/Drill?path=Local%2FNews
```

into

```
http://www.wirehose.com/Resources/Local/News
```

Most WireHose components which generate links to direct actions, such as WHTagDriller or

WHNavigationBox, provide a rewriteRule binding. These bindings are usually resolved via the layout dictionary.

Special components

WireHose includes a lot of reusable components. Here are a few of our favorites that weren't used in the Hello World tutorial:

WHShowTagDataSource

Allows embedding a WHTagDataSource in a web page, through bindings such as optionalTags and keywordString. You can set the optionalTags and requiredTags bindings to an individual tag, arrays of tags, or a string or arrays of strings which will be interpreted as tagpaths.

WHHTMLString

Renders HTML text, substituting for `<WIREHOSE type=FETCHER tag=`*`tagPath`* `fetchlimit=`*`numItemsToDisplay`*`>` with the rendered objects matched by the specified tag.

WHMatrixTable

WHMatrixTable displays an array of objects in a multi−column layout. It attempts to be smart about the number of columns and rows it uses to render itself; this behavior is controlled by several optional bindings. If all these are left unbound, WHMatrixTable will set its rows & columns roughly proportional to the "golden ratio".

Caching

WHCachingDataSource

WHCachingDataSource is an abstract class which provides all the necessary infrastructure for fetching enterprise objects into a cache, returning subsets of objects from the cache, and invalidating the cache when necessary.

If a caching datasource is created with an owner which implements the WHFetcher interface, such as a WHTagFetcher, it will use its owner's values for its properties such as `fetchLimit`. You can also instantiate a caching datasource and set the its properties directly; this is useful when using a caching datasource as an EODataSource for a display group, for example.

WireHose provides two concrete implementations, WHTagDataSource and WHQualifierDataSource. You can create your own by implementing `fetchResourcesIntoEditingContext`, which will be called as necessary by WHCachingDataSource itself.

WHConcreteFetcher

WHConcreteFetcher provides an abstract implementation of a channel which owns a caching datasource. WireHose provides two concrete implementations, WHTagFetcher and WHQualifierFetcher. WHConcreteFetcher is a full−service implementation of the WHFetcher interface, and requires minimal customization to handle new fetching behavior.

ShouldInvalidateCache notifications

To prevent caches from becoming stale, WHFetcher defines the ShouldInvalidateCache notification. Fetchers whose cache should be invalidated when a particular object changes can register for these notifications. For example, WHTagDataSource registers for ShouldInvalidateCache notifications for each of its optional and required tags, and will invalidate its cache if any of its tags are changed.

For fetchers that deal with enterprise objects, the notification object will be the globalID of the object which has changed; WHTag and the default implementation of the taggable interface will automatically post these notifications for tags.

WHConcreteFetcher and its companion WHCachingDataSource provide an implementation of the WHFetcher interface for fetching enterprise objects; they handle caching and propagation of ShouldInvalidateCache notifications.

Multiple affiliates

WireHose has special support for creating and deploying large numbers of re−branded portals sharing some common resources, as for example in an application service provider environment, community publishing, or higher education.

WireHose provides built−in support for handling multiple sites from a single set of databases via subentities. Most WireHose base entities, such as WHTag, WHTagFetcherFactory, WHUser, etc., have an `affiliate` property, which is used to identify to which affiliate a particular object belongs. The current affiliate name is controlled by the `WHDefaultAffiliate` property

For large deployments, WireHose can make extensive use of entity inheritance, taking advantage of the fact if an entity is not visible at runtime, any database rows described that entity are simply unavailable to the application. This is a simple but effective way to partition objects between separate application instances which share identical codebases and differ only in configuration files or launch arguments.

For example, if you are deploying multiple news portals, users connecting to the Seattle portal should only see Seattle−area traffic cams, and Portland users should only see Portland−area traffic cams, but both should have access to national newsfeeds. WHEnterpriseObject provides several methods for dynamically creating subentities at runtime, so you don't have to manually model many common types of inheritance in EOModeler.

Affiliate−based inheritance

The most prevalent inheritance model used in WireHose is that of affiliate−based inheritance. This approach is used by WHTag, WHUser, WHTagFetcherFactory, WHDrillFetcherFactory and WHQualifierFetcherFactory, and can easily be used in your own resource and channel factory entities.

In this approach, you define an attribute on your base entity called "affiliate". This attribute will be used with a restricting qualifier to identify subentities. The restricting qualifier for the base entity is "affiliate = nil", and the restricting qualifier for an affiliate−based subentity would be "affiliate = *affiliateName*". For example, given a base entity named "Picture" and an affiliate of "Seattle", the subentity would be named "SeattlePicture" and the restricting qualifier would be "affiliate = 'Seattle'".

As an alternative to the affiliate attribute, you can use an attribute called "entityType". If the base entity has an attribute named entityType, the restricting qualifier would be "entityType = *entityName*". This is the approach used by WHTag.

Automatic subentity creation

If the system property WHDisableAutoSubEntities is false, WHApplicationHelper uses WHEnterpriseObject's `createSubEntitiesForAffiliates` to automatically create subentities for a given list of affiliates for each entity which has an "affiliate" or "entityType" attribute. You can override this behavior on a per−entity basis by including a `WHPreventAutoSubEntities = YES` entry in the entity's userinfo dictionary. (If you are not using the WireHoseLayoutSupport framework, as in a command−line tool, you will need to call createSubEntitiesForAffiliates yourself.)

You can also create subentities at runtime which use multiple−table rather than single−table inheritance. WHEnterpriseObject's `createSubEntity` method lets you specify a restricting qualifier as well as an external name (i.e., table name) for a subentity. Any subentity created by WHEnterpriseObject will have a `WHCreatedSubEntity = YES` entry in its userinfo dictionary, and `createSubEntitiesForAffiliates` will not create subentities for a given entity if it finds this. WHApplicationHelper posts a `notification` before it creates subentities, so you can register for this notification if you

need to customize subentity creation.

The property `WHUserEntityName` controls which base entity WHApplicationHelper will use when fetching and creating users. The actual entity fetched and created will be an affiliate−based subentity of this entity for the default affiliate, if available. For example, if `WHDefaultAffiliate` is "Seattle" and `WHUserEntityName` is "WHUser" (the default), users and guest users will be of the "SeattleUser" entity if it exists.

You can override WHApplicationHelper's automatic subentity creation in several ways:

- If the property `WHDisableAutoSubEntities` is YES (or true), WHApplicationHelper will not create any subentities during startup.
- Since WHEnterpriseObject's subentity creation methods won't create a subentity if it already exists, you can create subentities in response to the `ApplicationHelperWillFinishInitializing` notification, which is posted before WHApplicationHelper creates any subentities.
- Add entries to the "autoSubEntities" key in your application's layout dictionary, which will get created before WHApplicationHelper creates subentities for all available affiliates. Each key in this dictionary is the name of a subentity to create, and its associated value is a dictionary describing the subentity. For example:

```
MySpecialPicture = {
     externalName = Picture;
     restrictingQualifier = "(affiliate = 'MySpecial')";
     parent = Picture;
 };
```

Depending on how you have your inheritance set up, you can specify one or both of externalName and restrictingQualifier.

Multiple affiliate best practices

The best way to handle multiple affiliates in your code is to not make any assumptions about whether or not subentities are available. This section provides some techniques you should follow in your own code.

Modeling entities

To model an entity which may become the parent of affiliate−based subentities, set the entity so it is not abstract and does not have a restricting qualifier on the affiliate property. WireHose will add the appropriate restricting qualifier to the parent entity when creating the first subentity.

Creating objects

WHEnterpriseObject provides the `createAndInsertInstance(EOEditingContext, String entityName, String affiliateName)` utility method, which functions similarly to the one defined in EOUtilities. The WireHose version provides an additional parameter, affiliateName, which is used to specify the affiliate the newly created instance should belong to.

If there is an affiliate−based subentity available for the specified entity, then the returned object will be of that entity, otherwise it will be of the specified entity. In either case, the affiliate property will be set on the newly created object if the entity has a class property named "affiliate".

Fetching objects

If you are fetching objects which belong to a specific affiliate, check for the existence of an affiliate−based subentity like this:

```
EOEditingContext ec; // assume exists
EOFetchSpecification fetchSpec; // assume exists
String affiliateName = "MyAffiliate";
String entityName = "SomeEntity";
String entityToFetch =
        WHEnterpriseObject.subEntityNameForAffiliate(entityName, affiliateName);
if (EOUtilities.modelGroup(ec).entityNamed(entityToFetch) == null) {
        entityToFetch = entityName;
}
fetchSpec.setEntity(entityToFetch);
NSArray objs = ec.objectsWithFetchSpecification(fetchSpec);
```