This page last changed on Apr 20, 2010 by shivakumar.

Apache Wink 1.1

Apache Wink is a complete Java based solution for implementing and consuming REST based Web Services. The goal of the Apache Wink framework is to provide a reusable and extendable set of classes and interfaces that will serve as a foundation on which a developer can efficiently construct applications.

Contents

1 Introduction to Apache Wink

This page last changed on Apr 20, 2010 by bluk.

Introduction to Apache Wink

Apache Wink 1.1 is a complete Java based solution for implementing and consuming REST based Web Services. The goal of the Wink framework is to provide a reusable and extendable set of classes and interfaces that will serve as a foundation on which a developer can efficiently construct applications.

Wink consists of a Server module for developing REST services, and of a Client module for consuming REST services. It cleanly separates the low-level protocol aspects from the application aspects. Therefore, in order to implement and consume REST Web Services the developer only needs to focus on the application business logic and not on the low-level technical details.

The Wink Developer Guide provides the developer with a rudimentary understanding of the Wink framework and the building blocks that comprise it.

Welcome to Apache Wink

Wink is a framework for the simple implementation and consumption of REST web services. REST is an acronym that stands for REpresentational State Transfer. REST web services are "Resources" that are identified by unique URIs. These resources are accessed and manipulated using a set of "Uniform methods". Each resource has one or more "Representations" that are transferred between the client and the service during a web service invocation.

The central features that distinguish the REST architectural style from other network-based styles is its emphasis on a uniform interface, multi representations and services introspection.

Wink facilitates the development and consumption of REST web services by providing the means for modeling the service according to the REST architectural style. Wink provides the necessary infrastructure for defining and implementing the resources, representations and uniform methods that comprise a service.

REST Architecture

For a detailed understanding of the REST architecture refer to the description by Roy Fielding in his dissertation, The Design of Network-based Software Architectures. In particular, Chapter 5 Representational State Transfer (REST) describes the principles of the architecture.

REST Web Service

Figure 1: REST Web service design structure

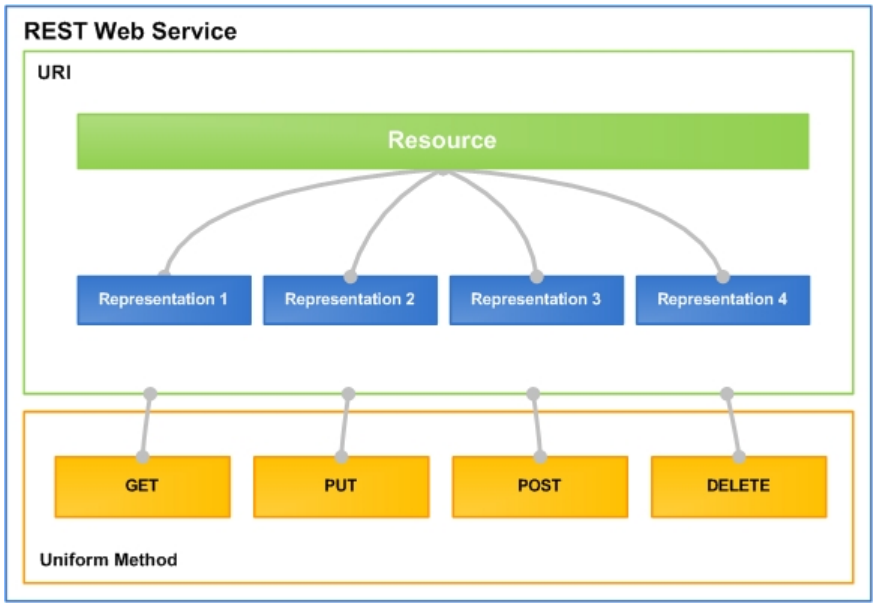


Figure 1 demonstrates the design principles and components that comprise a REST web service. Wink reflects these design principles in the implementation of web services.

Apache Wink Open Development

The purpose of this document is to provide detailed information about Wink 1.1 and describe the additional features that the Apache Wink 1.1 runtime provides in addition to the JAX-RS Java API for REST Web Service specification.

In addition to the features description, this document also provides information regarding implementation specific issues.

This document provides the developer with a rudimentary understanding of the Wink 1.1 framework in order to highlight the underlying concepts and precepts that make up the framework in order to create a basis for understanding, cooperation and open development of Wink.

JAX-RS Specification Document
For more information on the JAX-RS functionality, refer to the JAX-RS specification document, available at the following location:
http://jcp.org/aboutJava/communityprocess/final/jsr311/index.html

JAX-RS Compliancy

Apache Wink 1.1 aims to be a fully compliant implementation of the JAX-RS v1.1 specification.

JAX-RS is a Java based API for RESTful Web Services is a Java programming language API that provides support in creating web services according to the Representational State Transfer (REST) architectural style. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.

2 Apache Wink Building Blocks

This page last changed on Apr 20, 2010 by bluk.

Apache Wink Building Blocks

In order to take full advantage of Apache Wink, a basic understanding of the building blocks that comprise it and their functional integration is required. The following section provides an introduction to the basic concepts and building blocks of Apache Wink. In order to gain in-depth understandings of the building blocks refer to the table of contents where these concepts are expanded and addtional example are used to further illustrate the Apache Wink and JAX-RS SDK technologies.

This section contains the following topics:

Service Implementation Building Block Overview

As mentioned in the "**Apache Wink Introduction**" section, Apache Wink reflects the design principles of a REST web service. It does so by providing the developer with a set of java classes that enable the implementation of "**Resources**", "**Representations**" and the association between them. Wink also enables the developer to define the resource URI and the "**Uniform methods**" that are applicable to the resource.

Resource

A "**resource**" represents a serviceable component that enables for the retrieval and manipulation of data. A "**resource class**" is used to implement a resource by defining the "**resource methods**" that handle requests by implementing the business logic. A resource is bound or anchored to a URI space by annotating the resource class with the @Path annotation.

Providers

A provider is a class that is annotated with the @Provider annotation and implements one or more interfaces defined by the JAX-RS specification. Providers are not bound to any specific resource. The appropriate provider is automatically selected by the Apache Wink runtime according to the JAX-RS specification. Apache Wink supplies many providers, however, application developers may supply their own which take precedence over any provider in the default runtime.

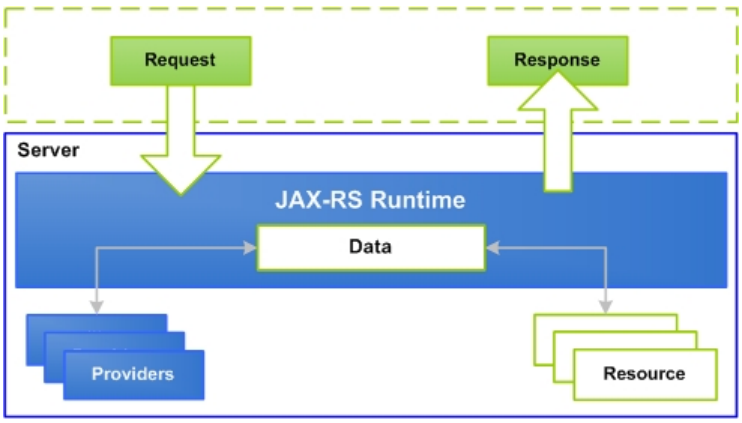
There are three types of providers defined by the JAX-RS specification:

- Entry Providers
- Context Providers
- Exception Mapping Provider

Entity Provider

An "**Entity Provider**" is a class that converts server data into a specific format requested by the client and/or converts a request transmitted by the client into server data. For instance, a String entity provider can turn request entities (message bodies) over the wire into a Java type (java.lang.String). Entity providers can also turn native Java types such as a java.lang.String into an appropriate response entity. An entity provider can be restricted to support a limited set of media types using the @javax.ws.rs.Produces and @javax.ws.rs.Consumes annotations. An entity provider is configured to handle a specific server data type by implementing the javax.ws.rs.ext.MessageBodyWriter and/or javax.ws.rs.ext.MessageBodyReader interfaces.

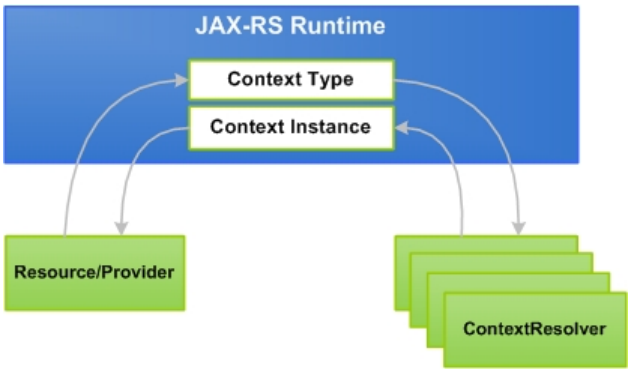
Figure 2: Entity Provider Diagram



Context Provider

Context providers are used to supply contexts to resource classes and other providers by implementing the javax.ws.rs.ext.ContextResolver interface. For example, if a custom JAXBContext is required to serialize or deserialize JAXB objects, an application can provide a ContextResolver that will return a specific instance of a JAXBContext. Context providers may restrict the media types that they support using the @javax.ws.rs.Produces annotation.

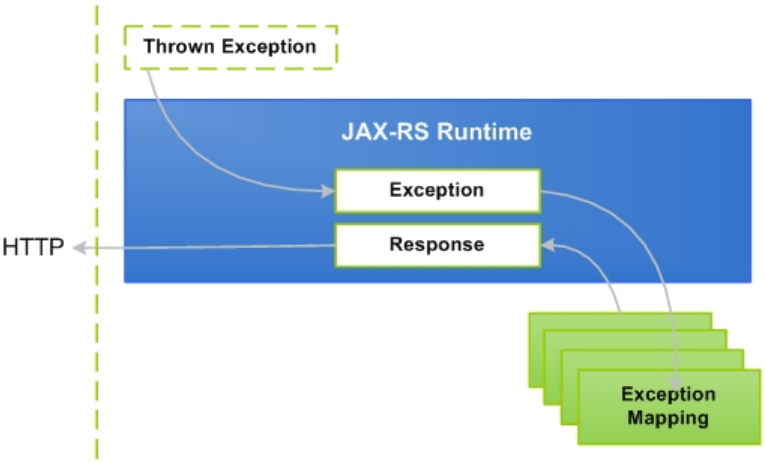
Figure 3: Context Provider Diagram



Exception Mapping Provider

Exception mapping providers map exceptions into an instance of a javax.ws.rs.core.Response by implementing the javax.ws.rs.ext.ExceptionMapper interface. The Response objects can contain response information such as status codes, HTTP headers, and a response entity (message body). When a resource method throws an exception, the runtime will try to find a suitable ExceptionMapper to "translate" the exception to an appropriate Response object.

Figure 4: Exception Mapping Provider Diagram

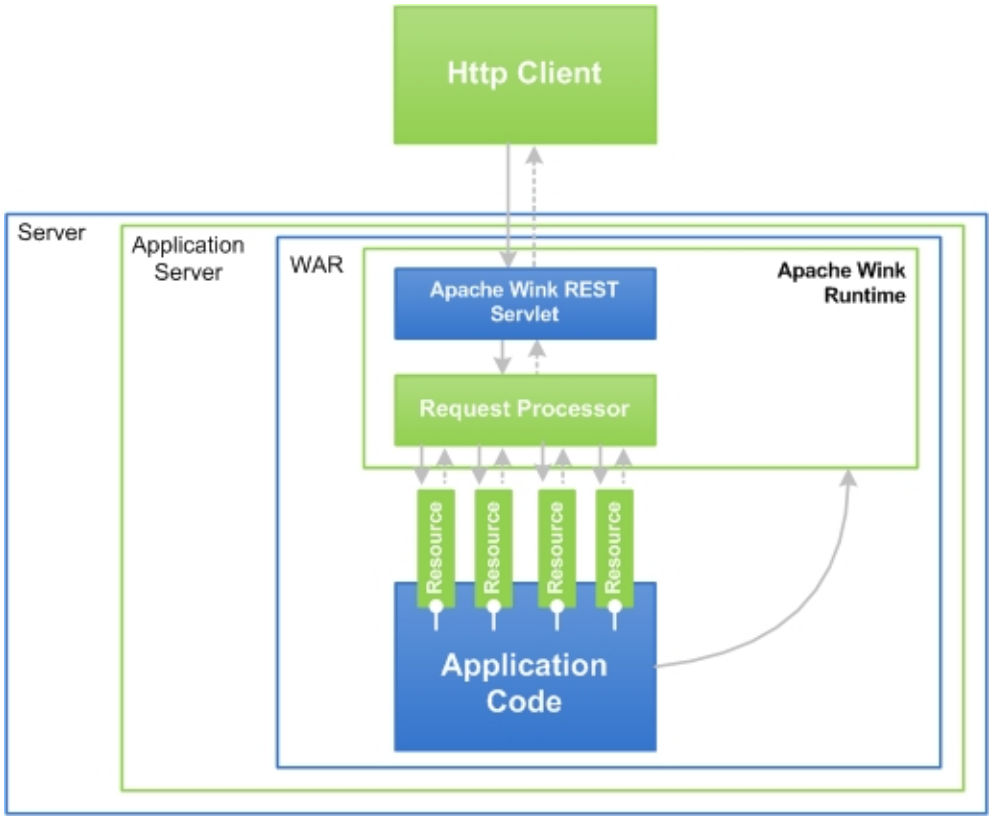


URI Dispatching

Designing an efficient REST web service requires that the application developer understands the resources that comprise the service, how to best identify the resources, and how they relate to one another.

RESTful resources are identified by URIs in most cases related resources have URIs that share common path elements.

Figure 5: Apache Wink Logic Flow



Apache Wink Logic Flow

Figure 5 illustrates the Apache Wink logic flow. The HTTP request sent by the client invokes the "**Apache Wink REST Servlet**". The REST servlet uses the "**Request Processor**" and the request URI in order to find, match and invoke the correct resource method.

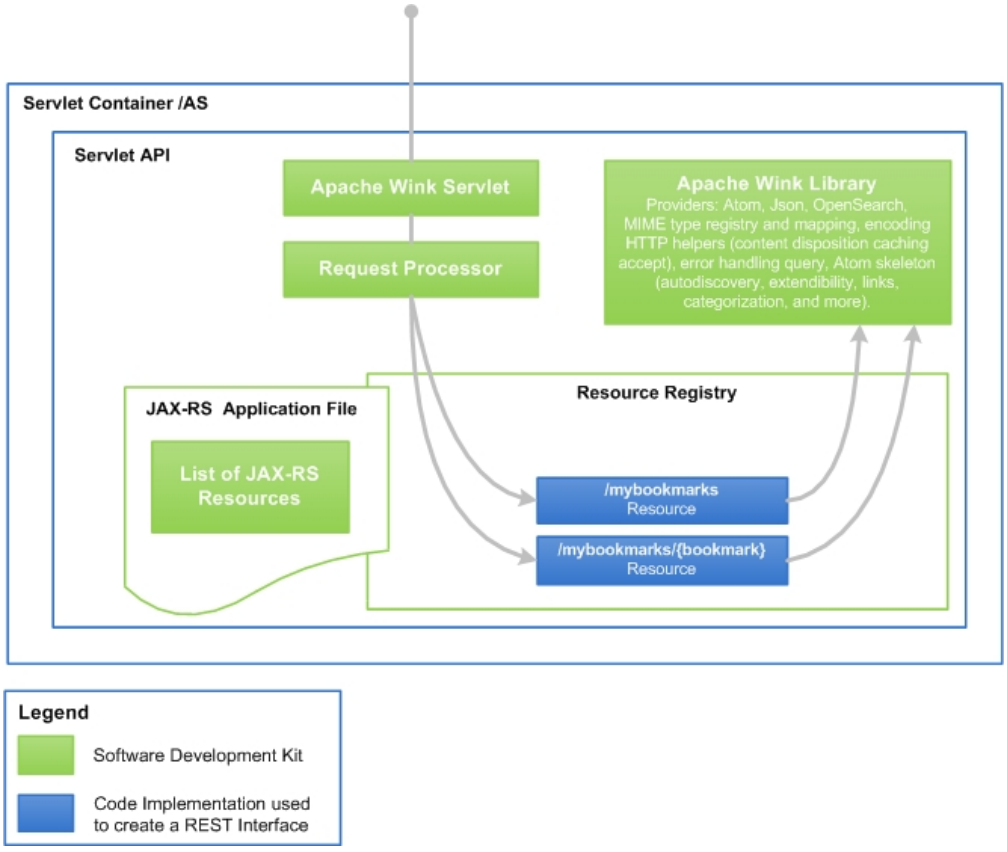
Bookmarks Example

Throughout this document, various project examples are used in order to describe the functionality and processes that comprise the Apache Wink.
In order to explain the REST design principles used in the Apache Wink this developer guide refers to the "**Bookmark**" example project found in the examples folder located in the Apache Wink distribution.

Refer to the code (using an IDE application) in the example in conjunction with the following explanations and illustrations in this developer guide.

Apache Wink Servlet and Request Processor

Figure 6: Apache Wink REST Servlet and Request Processor for the Bookmark Services



Server and Request Processor

Figure 6 shows the Apache Wink servlet and request Processor concept in the context of the application server. In the "**Bookmarks**" example in Figure 6 there are two Resources, the first Resource is associated with the **/mybookmarks** URI and manages the bookmarks collection, the second resource is associated with the **/mybookmarks/{bookmark}** Resources and manages an individual bookmark within the collection.

The Resources' defined by the web service and managed by Apache Wink are referred to as **"URI space"**. The Resources space is the collection of all URI's that exist in the same context. Figure 6 shows the URI space that contains **/mybookmarks** and **/mybookmarks/{bookmarks}**.

URI Space

The Bookmarks service URI space consists of the following URI space items and detailed descriptions about their context and functionality.
Table 1: URI Management

URI space Item	Description
/Bookmark/rest	This URI is the root context of the bookmark service and the entry point of the URI space of the service. An HTTP GET request to this URI returns a "Service Document" which is automatically generated by Apache Wink. The service document provides information about all available collections in the URI space.
/Bookmark/rest /mybookmarks	This URI is associated with a collection of bookmarks resources. Clients use the HTTP GET method in order to retrieve a representation of

	the collection and HTTP POST method in order to create a new item in the collection.
/Bookmark/rest /mybookmarks/ {bookmark}	This URI template is associated with a single bookmark resource. Clients use the HTTP GET method in order to retrieve a representation of the resource, HTTP PUT method is used in order to update the resource and HTTP DELETE method is used in order to delete the resource.

Assets

Assets are classes that contain "**web service business logic**" implemented by the developer. Each Asset is associated with one or more URI. The Apache Wink dispatcher invokes the Asset, which is associated with the URI found in the HTTP request.

An Asset class can implement one or more methods, each method is associated with a single HTTP method (GET, HEAD, POST, PUT, DELETE etc). The methods can be associated with a MIME type of a produced representation. Methods that handle HTTP verbs of requests with a body (such as PUT, POST) are also associated with the MIME type of the HTTP request.
The Asset class can be registered to the Apache Wink using the "Spring context xml" or by using a registration API.

Spring Context Configuration
For further information regarding the Spring Context, refer to 5.5 Spring Integration in section 5 Apache Wink Server.

Annotations

Annotations are a special text notations, or metadata, added to Java version 1.5. Annotations in Java source code can affect both compilation and runtime behavior of the resulting Java classes.
JAX-RS is implemented by the use of annotations that are defined in the JAX-RS specification. Apache Wink provides a set of additional annotations that enrich the functionality of the JAX-RS enabled application.

The following table describes the additional Apache Wink annotations:

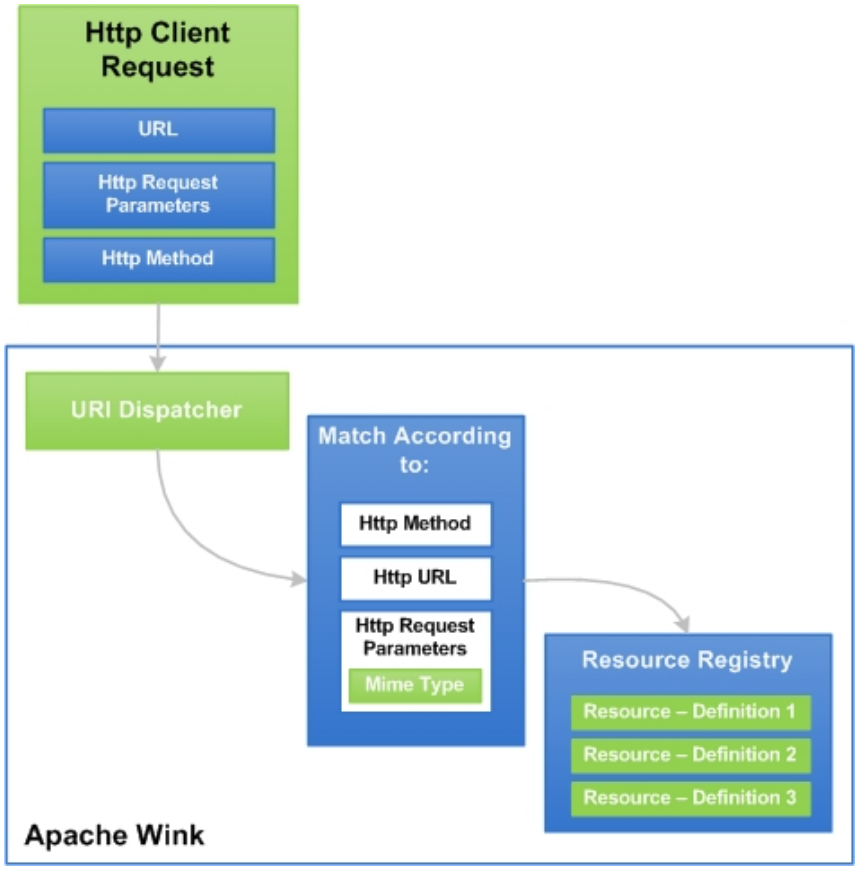
Annotation	Precedes	Description
@Workspace	Resource	Associate a "Collection Resource" with a workspace element and collection elements in an APP Service Document
@Scope	Resource /Provider	Defines the default lifecycle behavior for resources and providers, and the option for controlling the lifecycle through the javax.ws.rs.core.Application class
@Parent	Resource	Provides the ability to define a base template URI for the URI specified in a resources @Path annotation

@Asset	Class	Used by the Apache Wink runtime in order to identify an entity as an Asset

URL Handling

The Apache Wink receives HTTP requests and then dispatches a wrapped HTTP request to the appropriate Resource method.
The HTTP request is match to the Resource method based on the HTTP request parameters, the Resource method definitions and the MIME type.

Figure 7: URL Request Handling



Request Handling

Figure 7 demonstrates the HTTP Client request path to the URI dispatcher, once the dispatcher receives the request it is then matched according to the HTTP method, URL and MIME type and finally the Resource registry definition.

HTTP Methods - GET, POST, PUT, DELETE and OPTIONS

The common HTTP 1.1 methods for the Apache Wink are defined in the following section. This set of methods can be expanded.

Method Usage

Table 3: HTTP Methods

Method	Safe	Idempotent	Cacheable
GET	X	X	X
HEAD	X	X	X
PUT		X	
POST			*
DELETE		X	
OPTIONS			

Key - X

- **Safe** - does not affect the server state
- **Idempotent** - a repeated application of the same method has the same effect as a single application
- **Cacheable** - a response to a method is cacheable if it meets the requirements for HTTP caching
- ***** - Responses to this method are not cacheable, unless the response includes an appropriate Cache-Control or Expires header fields. However, the 303 response can be used to direct the user agent to retrieve a cacheable resource.

GET

The GET method is used to retrieve information from a specified URI and is assumed to be a safe and repeatable operation by browsers, caches and other HTTP aware components. This means that the operation must have no side effects and GET method requests can be re-issued.

HEAD

The HEAD method is the same as the GET method except for the fact that the HEAD does not contain message body.

POST

The POST method is used for operations that have side effects and cannot be safely repeated. For example, transferring money from one bank account to another has side effects and should not be repeated without explicit approval by the user.

The POST method submits data to be processed, for example, from an HTML form, to the identified resource. The data is included in the body of the request. This may result in the creation of a new resource or the updates of existing resources or both.

PUT

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one residing on the origin server.

If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI.

DELETE

The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method can be overridden on the origin server.
The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully.

OPTIONS

The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI.
This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval.

Basic URL Query Parameters

A URL parameter is a name and value pair appended to a URL. The parameter begins with a question mark "**?**" and takes the form of name=value.
If more than one URL parameter exists, each parameter is separated by an ampersand "**&**" symbol. URL parameters enable the client to send data as part of the URL to the server.

When a server receives a request and parameters are appended to the URL of the request, the server uses these parameters as if they were sent as part of the request body. There are several predefined URL parameters recognized by Apache Wink when using Wink providers. The following table lists the parameters commonly used in web service URLs. These special URL parameters are defined in the "**RestConstants**" class.

Query Parameters

Table 4: URL Parameters

Parameter	Description	Value
alt	Provides an alternative representation of the specified MIME type. Apache Wink recognizes this as a representation request of the highest priority.	MIME type, e.g. "text%2Fplain"
absolute-urls	Indicates to Apache Wink that the generated links in the response should be absolute, mutual exclusive with the relative-urls parameter	NONE
relative-urls	Indicates to Apache Wink that the generated links in the response should be relative, mutual exclusive with the absolute-urls parameter	NONE
callback	Wrap javascript representation in callback function, is relevant when requested with an application/json MIME type.	name of callback function. For example, "myfunc"

Combining URL Parameters

A single URL can contain more than one URL parameter, example **"?alt=text%2Fjavascript &callback=myfunc"**(where "%2F" represents escaped "/").

Apache Wink Building Blocks Summary

The previous section "**Service Implementation Building Blocks**" outlines the basic precepts and building blocks that comprise the service side of Apache Wink.
In order to understand the relationship between the building blocks that comprise Apache Wink a set of example applications have been designed and built that provide a reference point that demonstrate a rudimentary understanding about the functionality of Apache Wink.

Apache Wink Examples

The following examples applications are used in this "**Apache Wink Developer Guide**".

- Bookmarks
- HelloWorld
- QADefects

Bookmarks Project

This developer guide uses the bookmarks example application in order to describe the logic flow process within Apache Wink.
Refer to the comments located in the "**Bookmarks**" example application code for in-depth explanations about the methods used to build the bookmarks application.

HelloWorld Project

Complete the step-by-step "**HelloWorld**" tutorial in chapter 3 "**Getting Started with Apache Wink**" and then follow the installation instructions on page xx, in order to view the "**Bookmarks**" example application from within the Eclipse IDE.

QADefects

The QADefects example application illustrates the advanced functionality of Apache Wink by implementing most of the features provided by the Apache Wink (Runtime) framework.

Apache Wink Client Component Basics Overview

The Apache Wink Client interacts with REST Web-Services. It maps REST concepts to Java classes and encapsulates underlying REST related standards and protocols, as well as providing a plug-in mechanism for raw HTTP streaming data manipulation. This mechanism also provides the ability to embed cross application functionality on the client side, such as security, compression and caching.

Figure 8: Apache Wink Client Simplified Breakdown

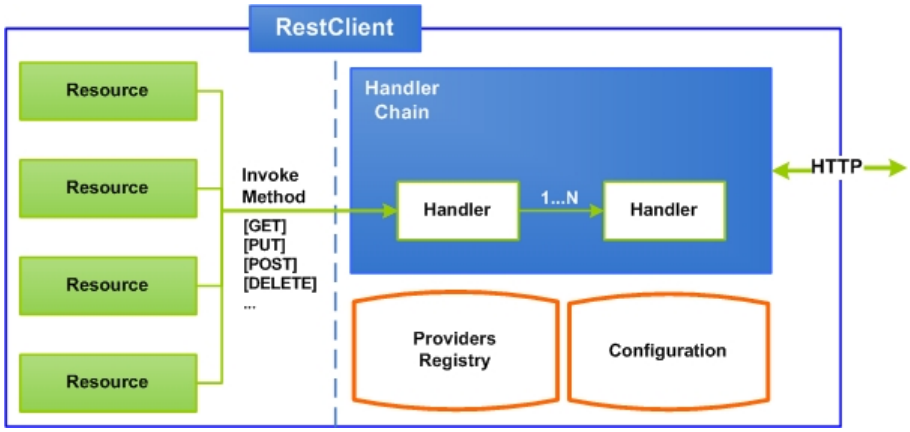


Figure 8: The illustration shows the basic elements that comprise the Apache Wink Client. The Apache Wink Client utilizes the providers mechanism defined by the JAX-RS specification to perform reading and writing of java objects. The Apache Wink Client is pre-initialized with the same providers that are predefined by the Apache Wink JAX-RS server implementation.

Apache Wink Client Components

The Apache Wink Client is comprised of several key elements that together create a simple and convenient framework for the consumption of REST based web services. The client is an abstraction of REST concepts modeled into simple java classes that encapsulate the underlying HTTP protocol, used for the service invocation.

The Apache Wink Client uses the java HttpURLConnection class for the HTTP invocation. The Apache Wink Client also provides a module that utilizes the Apache HTTP Client instead of the default HttpURLConnection class.

The following section provides an overview of the key elements and classes that comprise the Apache Wink Client.

RestClient Class

The RestClient class is the central access point to Apache Wink Client. It provides the user with functionality that enables the creation of new Resource instances. The RestClient provides the user with the ability to set different configuration parameters and custom JAX-RS providers and propagates them to the created resources.

Resource Interface

The Resource interface represents a single web resource located at a specific URL, enabling for the manipulation and retrieval of the resource by the invocation of different HTTP methods on the resource instance. The resource interface is implemented following the Builder design pattern so method calls can be easily aggregated and in order to enable easy construction of requests and setting of the resource properties prior to invocation.

ClientRequest Interface

The ClientRequest interface represents a request issued by invoking any one of the invocation methods on a Resource. An instance of a ClientRequest is created at the beginning of an invocation and passed to all the client handlers defined on the client that was used for the invocation.

ClientResponse Interface

The ClientResponse interface represents an HTTP response that is received after invoking any one of the invocation methods on a Resource. An instance of a ClientResponse is created by the ConnectionHandler at the end of the handler chain, and is returned from every handler on the chain.

ClientConfig Class

The ClientConfig class provides client configuration when instantiating a new RestClient. The ClientConfig is implemented using the Builder design pattern so method calls can be easily aggregated. Custom Providers and client Handlers are set on the ClientConfig instance prior to the creation of the RestClient.

ClientHandler Interface

Client handlers are used to incorporate cross invocation functionality. The ClientHandler interface is implemented by client handlers, and the handle() method is invoked for every request invocation in order to allow the handler to perform custom actions during the request invocation.

InputStreamAdapter Interface

The InputStreamAdapter interface is used to wrap the response input stream with another input stream in order to allow the manipulation of the response entity stream. The adapt() method is called after reading the response status code and response headers, and before returning to the ClientResponse to the handlers on the chain.

OutputStreamAdapter Interface

The OutputStreamAdapter interface is used to wrap the request output stream with another output stream to allow the manipulation of the request entity stream. The adapt() method is called before writing the request headers to allow the adapter to manipulate the request.

EntityType Class

The EntityType is used to specify the class type and the generic type of responses. Typically, an anonymous "**EntityType**" instance is created in order to specify the response type, as is shown in the following code example:

```
Resource resource = client.resource(uri);
List<String> list = resource.get(new EntityType<List<String>>() {});
```

ApacheHttpClientConfig Class

The "**ApacheHttpClientConfig**" Configuration object configures the Apache Wink Client to use the Apache HttpClient as the underlying HTTP client. The following code snippet, demonstrates the typical usage:

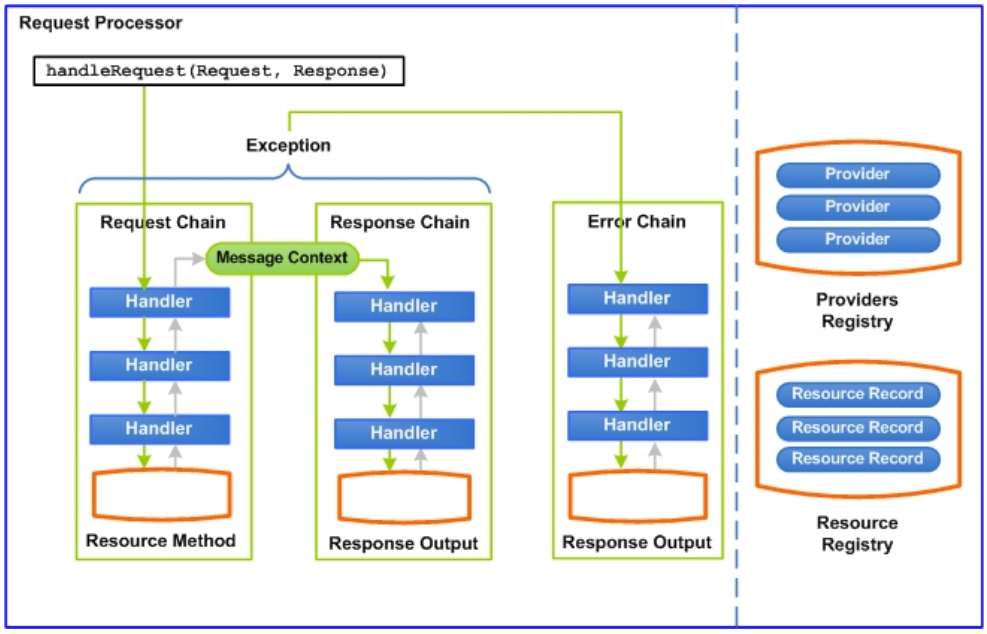
```
// creates the client that uses Apache DefaultHttpClient as the underlying Http client.
RestClient client = new RestClient(new ApacheHttpClientConfig(new DefaultHttpClient()));

// creates the resource
Resource resource = client.resource("http://myhost:80/my/service");

// invokes a GET method on the resource and receives the response entity as a string
String entity = resource.get(String.class);
```

```
...
```

The Apache Wink Runtime

The Apache Wink runtime is deployed on a JEE environment and is configured by defining the RestServlet in the web.xml file of the application. This servlet is the entry point of all the HTTP requests targeted for web services, and passes the request and response instances to the Wink engine for processing.

Figure 9: Apache Wink Request Processor Architecture



The diagram illustrates the core components of the Apache Wink runtime. The Wink engine is the RequestProcessor. It builds an instance of a MessageContext with all of the required information for the request and passes it through the engine handler chains. The handler chains are responsible for serving the request, invoking the required resource method and finally generating a response.
In case of an error, the RequestProcessor invokes the Error chain with the generated exception for producing the appropriate response.
The Apache Wink runtime maintains providers and resources in two registries, the "**providers registry**" and the "resource registry" utilizing them during request processing.

Request Processor

The RequestProcessor is the Apache Wink engine, that is initialized by the RestServlet and is populated with an instance of a DeploymentConfiguration.
When a request is passed to the handleRequest() method of the RequestProcessor, a new instance of a MessageContext is created.
The MessageContext contains all of the information that is required for the Wink runtime to handle the request. The RequestProcessor first invokes the Request Handler Chain and then the Response Handler Chain.

If an exception occurs during any stage of the request processing, the RequestProcessor invokes the Error Handler Chain for processing the exception.

Deployment Configuration

The Apache Wink runtime is initialized with an instance of a Deployment Configuration. The Deployment Configuration holds the runtime configuration, including the handler chains, registries, configuration properties.
The Deployment Configuration is initialized with an instance of a JAX-RS Application used for obtaining user resources and providers.

Customization of the Handlers Chain

The handler chain is customized by extending the org.apache.wink.server.handlers.HandlersFactory class, overriding specific methods and specifying the new class in the web.xml file of the application.
In order to specify a different HandlersFactory class instead of the default handlers, specify an init parameter for a custom properties file to be loaded by the RestServlet. Then, the value of the wink.handlersFactoryClass property must be set as the fully qualified name of the customized handlers class in the properties file.

```
<servlet>
 <servlet-name>restSdkService</servlet-name>
 <servlet-class>
 org.apache.wink.server.internal.servlet.RestServlet
 </servlet-class>
 <init-param>
 <param-name>propertiesLocation</param-name>
 <param-value>path/to/my-wink-properties.properties</param-value>
 </init-param>
</servlet>
```

In the my-wink-properties properties file:

```
wink.handlersFactoryClass=org.apache.wink.MyCustomHandlersFactory
```

See the JavaDoc for the HandlersFactory API.

Handler Chains

The handler chain pattern is used by the Wink runtime for implementing the core functionalities.
There are three handler chains utilized by the Wink runtime:

- RequestHandlersChain
- ResponseHandlersChain
- ErrorHandlersChain

Handler Chains
For further information regarding the "**Handler Chains**", refer to section 5 **Apache Wink Server**, 5.7 Handler Chain - Runtime Extension

---

Registries

The Apache Wink runtime utilizes two registries for maintaining the JAX-RS resources and providers. Both registries maintain their elements in a sorted state according to the JAX-RS specification for increasing performance during request processing. In addition to the JAX-RS specification sorting, Wink supports the prioritization of resources and providers.

> **Resources and Providers Prioritization**
> For further information regarding *"**Resources and Providers Prioritization**", refer to the section 5.1 Registration and Configuration.

Resource Registry

Firgure 10: Resource Registry Architecture

The resources registry maintains all of the root resources in the form of Resource Records.
A Resource Record holds the following:

- **URI Template Processor** - represents a URI template associated with a resource. Used during the resource matching process.
- **Resource Metadata** - holds the resource metadata collected from the resource annotations.
- **Sub-Resource Records** - records of all the sub-resources (methods and locators) collected from the sub-resource annotations.
- **Resource Factory** - a factory that retrieves an instance of the resource in accordance to the creation method defined for the resource.
  Possible creation methods include:
  - - singleton
    - prototype
    - spring configuration
    - user customizable

3 Getting Started with Apache Wink

Getting Started with Apache Wink

Apache Wink consists of a main library and an additional set of dependant libraries. The Apache Wink distribution also includes an "**examples**" folder that contains all of the "**example projects**" referred to in this developer guide.

This section contains the following:

Getting Started with Apache Wink Overview

In order to gain an in-depth understanding of the Apache Wink SDK a series of example projects have been designed to help the developer clarify the underlying principles and usage of the Apache Wink.

Apache Wink Distribution Files

Get a version of Apache Wink from the web site (http://incubator.apache.org/wink/downloads.html). There are several JARs in the distribution but not all of them are required.

/dist directory

The main Apache Wink JARs are located in the "dist" directory:

- wink-common-<version>.jar
- wink-server-<version>.jar
- wink-client-<version>.jar

are the 3 main JARs. wink-common contains code that is required on both the server and client side. So if you only want to work with the server side code, you can use the wink-common and wink-server JARs. If you only want to build a client, you only need wink-common and wink-client.

- wink-<version>.jar

is also provided as a convenient JAR that includes all of the Apache Wink code in one comprehensive JAR.

/lib directory

The lib directory contains JARs that may be required to use Apache Wink depending on your environment. You will need the jsr311-api which contains the JSR-311 spec API classes.

You will likely need to have SLF4J in the classpath as it is the logging framework used in Apache Wink. SLF4J is a facade which allows logging messages to be directed to many popular logging frameworks. See SLF4J's website for more information on which SLF4J implementation you may need.

If you are running in an environment that does not have JAXB (i.e. Java 5), then you will also need the JAXB libraries.

/ext directory

Extensions to Apache Wink are located in the "ext" directory. You can add the appropriate extension to expand the capabilities.

/examples directory

The examples directory contains sample code that demonstrates various features of the server and client runtime.

Getting started with an Apache Wink server application

See 4.1 JAX-RS Getting Started for information on how to write a Hello World JAX-RS application.

Getting started with an Apache Wink client application

See 6.1 Getting Started with Apache Wink Client for information on how to write a client.

This page last changed on Apr 20, 2010 by bluk.

JAX-RS Concepts

JAX-RS (JSR 311) is the latest JCP specification that provides a Java based API for REST Web services over the HTTP protocol. The JAX-RS specification is an annotation based server side API.

Applying JAX-RS to Apache Wink

Apache Wink is an implementation of the JAX-RS specification, providing a rich set of features and expansions that extend and supplement the JAX-RS specification. Apache Wink is designed to be an easy to use, production quality and efficient implementation.

The Apache Wink architecture enables the incorporation of custom functionality via the use of handlers that provide for the manipulation of requests, Apache Wink also provides a powerful client module for the consumption of REST web services and is bundled with a range of built-in Providers that enable the developer to quickly implement applications that make use of industry standard formats such as XML, ATOM, APP, RSS, JSON, CSV, HTML.

Developing REST Applications

For those new to JAX-RS or developing REST applications, follow this in-progress developer guide. If you have comments or questions, send an e-mail to the appropriate Apache Wink mailing lists.

While this developer guide covers common scenarios, it is not intended to detail the JAX-RS specification itself. Please read the (brief yet understandable) specification if it is available to you.

Getting Started - A Simple JAX-RS Application

- Creating a Resource
- Creating a javax.ws.rs.core.Application sub-class
- Packaging Apache Wink with a Web Application
- Installation and Running the Application

Application Configuration

- javax.ws.rs.core.Application subclass
- Container environment information

Resources, HTTP Methods, and Paths

- Root Resource Methods
- javax.ws.rs.core.Response
- HTTP Methods
- Subresource Methods
- Subresource Locators
- Regular Expressions

Request and Response Entities (Message Bodies) and Media Types

- Produces/Consumes annotations
- ATOM
- XML
- JSON

- Standard Entity Parameter Types
- Custom application provided Message Body Readers/Writers
- Transfer Encoding
- Content Encoding

Parameters

- Query Parameters
- Path Parameters
- Matrix Parameters
- Header Parameters
- Cookie Parameters

HTTP Headers

- Common HTTP Request Headers
- Common HTTP Response Headers

Content Negotiation

- Using URIs to identify content type
- Using parameters to identify content type
- Using Accept headers

Using Request Context Information

- HTTP Headers
- URI Information
- Security Information
- Request
- Providers

Caching

- Expires
- Cache-Control

JAX-RS Application Configuration

This page last changed on Sep 10, 2009 by michael.

Applications

A JAX-RS application consists of one or more resources and zero or more providers. A JAX-RS application is packaged as a Web application in a .war file. The Application subclass, resource classes, and providers are packaged in WEB-INF/classes file and the required libraries are packaged in WEB-INF/lib. Included libraries can also contain resource classes and providers as desired.
When using a Servlet 3 container, the Application subclass is optional. By default, all root resource classes and providers packaged in the web application must be included in the published JAX-RS application.

Including Subclasses

An Application subclass can be included in a .war file to override the default behavior. If both getClasses and getSingletons return an empty list then the default "willset" of classes must be used. The implementations should support the Servlet 3 framework pluggability mechanism to enable portability between containers and to avail themselves of container-supplied class scanning facilities.

Servlet Containers

When using a non-JAX-RS aware servlet container, the servlet-class or filter-class element of the web.xml descriptor should name the JAX-RS implementation-supplied Servlet or Filter class respectively. The application-supplied subclass of the application is identified, using an init-param with a param-name of the "javax.ws.rs.application".

This page last changed on Oct 11, 2009 by michael.

JAX-RS Caching

TBD

Expires

TBD

Cache Control

TBD

This page last changed on Apr 20, 2010 by bluk.

Creating a Simple "Hello World" Application

The following example project will produce a simple JAX-RS application that can respond to requests at **"/helloworld"** with a **"Hello world!"** plain text resource. While not entirely RESTful, this example project is to show how to create a simple application and how to package it for consumption in a web container.

The application is packaged in a WAR file (which is similar to a JAR/ZIP file, except with special files in certain locations and a defined layout). It can be deployed in any web container, for example: Jetty, Tomcat and Geronimo. Perform the following steps in order to create the "**helloworld**" example application.

Step 1 - Creating the Root Resource

First, create a resource that will be used for HTTP GET requests to "**/helloworld**".

```
package org.apache.wink.example.helloworld;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/helloworld")
public class HelloWorldResource {

    @GET
    public String getMessage() {
        return "Hello World!";
    }
}
```

As shown above, the Java class is just a plain old Java object that has JAX-RS annotations.

Step 2 - Creating a javax.ws.rs.core.Application sub-class

For non-JAX-RS aware web container environments (most environments are currently non JAX-RS aware), a j**avax.ws.rs.core.Application** sub-class needs to be created which returns sets of JAX-RS root resources and providers. In the following example, there is only one root resource that will need to be returned.

```
package org.apache.wink.example.helloworld;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.core.Application;

public class HelloWorldApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<Class<?>>();
        classes.add(HelloWorldResource.class);
```

```
        return classes;
    }

}
```

Compiling the classes

Using the Apache Wink distribution's JARs in the classpath, compile the two classes from the previous example.

Step 3 - Creating a web.xml file

Now create a web.xml deployment descriptor. The deployment descriptor details information about the web application in the WAR. In this case, it says that the Apache Wink JAX-RS servlet should be initialized with a HelloWorldApplication instance.

In addition, any requests that begin with /rest/ will be handled by the Apache Wink JAX-RS servlet. So, the request URL would be "/rest/helloworld" to reach the HelloWorld resource.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
    <display-name>Hello world Web Application</display-name>
    <servlet>
        <servlet-name>HelloWorldApp</servlet-name>
        <servlet-class>org.apache.wink.server.internal.servlet.RestServlet</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>org.apache.wink.example.helloworld.HelloWorldApplication</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldApp</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Step 4 - Packaging the web application into a WAR file

Layout the application as follows and create a WAR file from the base directory (the one before WEB-INF). Create a WAR by running "jar cvf helloworld-jaxrs.war *" from the base directory.

Not every JAR in the lib directory is necessary for all environments. Read the documentation for more information about the requirements.

```
WEB-INF/classes/org/apache/wink/example/helloworld/HelloWorldApplication.class
WEB-INF/classes/org/apache/wink/example/helloworld/HelloWorldResource.class
WEB-INF/lib/commons-lang-2.3.jar
WEB-INF/lib/jaxb-api-2.1.jar
WEB-INF/lib/jaxb-impl-2.1.4.jar
WEB-INF/lib/stax-api-1.0-2.jar
WEB-INF/lib/jsr311-api-1.1.jar
WEB-INF/lib/slf4j-api-1.5.11.jar
```

```
WEB-INF/lib/slf4j-jdk-1.5.11.jar
WEB-INF/lib/wink-common-<version #>.jar
WEB-INF/lib/wink-server-<version #>.jar
WEB-INF/web.xml
```

Step 5 - Installing the WAR file into your environment

Most web container environments will take a WAR file and deploy it without any further configuration required. However, note the "**Context Root**" of the web application, or change it as required.

The context paths combine as follows:
http://<hostname>/<web application context root>/<servlet url mapping path>/helloworld

If the environment deployed the WAR file to a context root of "/helloworldapp", then the following URL would be required to reach the HelloWorldResource.

http://<hostname>/helloworldapp/rest/helloworld

JAX-RS Resources, HTTP Methods, and Paths

This page last changed on Sep 07, 2009 by michael.

Resources

Resources are one of the fundamental concepts in REST. REST emphasizes the manipulation of resources rather than issuing function calls. Resources have unique identifiers. In HTTP terms, this means associating every resource with at least one URL.

In order to manipulate a resource, requests are made with a specific HTTP method. For instance, in order to retrieve a representation of a resource, an HTTP GET request to the resource's URL is issued. In order to create a new item in a collection, an HTTP POST can be used with the collection URL.

Application developers define resources and the HTTP methods in order to quickly manipulate them by using regular plain old Java objects and JAX-RS annotations.

Defining a Root Resource (@Path on Java class)

Developers can use POJOs to define a resource. Root resources have a **@Path** annotation at the class declaration level. JAX-RS matches an incoming request's URL with the **@Path** annotation on all of an application's root resources in order to determine which initial Java class will handle the request.

Root resources Java class instances are created per request by default.

> **Reference**
> Refer to the JAX-RS Application configuration topic for more information.

Resource classes have methods that are invoked when specific HTTP method requests are made, referred to as resource methods. In order to create Java methods that will be invoked with specific HTTP methods, a regular Java method must be implemented and annotated with one of the JAX-RS @HttpMethod annotated annotations (namely, @GET, @POST, @PUT, and @DELETE).

For example, if a resource is located at a "/welcome" URL, the following root resource is defined.

```
@Path("/welcome")
public class WelcomeMessage {
   private String welcomeMessage = "Hello world!";

   @GET
   public String returnWelcomeMessage() {
      return welcomeMessage;
   }
}
```

Any incoming GET request that has the URL of "/welcome" is handled by WelcomeMessage class's **returnWelcomeMessage()** method. A string is returned that represents the response body and is sent as the response payload in a HTTP 200 status response.

Using a javax.ws.rs.core.Response

In the previous GET resource method example, the response body is returned as a String. If a more complex response is requiredfor example, additional HTTP response headers or a different status code, a **javax.ws.rs.core.Response** should be used as the Java method's return type. By building a Response object, additional information can be returned to the client.

```
@Path("/welcome")
```

```
public class WelcomeMessage {
    private String welcomeMessage = "Hello world!";

    @GET
    public Response returnWelcomeMessage() {
        String responseEntity = welcomeMessage;
        return Response.status(299).entity(responseEntity).header("CustomHeader", "CustomValue").build();
    }
}
```

The previous example uses 299 as the status code, standard HTTP status codes should be used in order to help clients understand responses.When using Strings as the response entity, different Java types may be used for complex responses.

> **Reference**
> Refer to the Request/Response entities page for more details on how request/response entities are handled.

Using Common HTTP Methods (@GET, @POST, @PUT, @DELETE)

The four most common HTTP methods are GET, POST, PUT, and DELETE.

As shown in the previous example, an HTTP GET response to "/welcome" invokes the returnWelcomeMessage() Java method.In order to add a Java method that would be invoked when a HTTP PUT request is made to "/welcome", the following code should be added:

```
@Path("/welcome")
public class WelcomeMessage {
    private String welcomeMessage = "Hello world!";

    @GET
    public String returnWelcomeMessage() {
        return welcomeMessage;
    }

    @PUT
    public String updateWelcomeMessage(String aNewMessage) {
        welcomeMessage = aNewMessage;
    }
}
```

Notice that the updateWelcomeMessage has an unannotated parameter which represents an incoming request's body.

> **Reference**
> Refer to the Request/Response entities page for more details on how request/response entities are handled.

Subresource Methods (@Path and @GET, @POST, @PUT, @DELETE on a Java method)

Sometimes it is easier having a root resource resolve a generic URL path and to have **@Path** annotated methods further resolve the request. For instance, suppose that a HTTP GET to "/administrator" returned generic information about an administrator. However, sometimes it is better to return smaller bits or more detailed information about the resource using a slightly different URL identifier. Suppose that a HTTP GET to "/administrator/name" should return the name. Instead of creating many root resource classes

for each URL, you can have the root resource initially resolve the beginning of the URL request and then further resolve the request against subresource methods.

Subresource methods are Java methods with a **@Path** annotation and a @HttpMethod annotated annotation (@GET, @POST, @PUT, @DELETE).

```
@Path("/administrator")
public class Administrator{

    @GET
    public String findUserInfo() {
        String userInfo = null;
        /* build user info */
        return userInfo;
    }

    @GET
    @Path("/name")
    public String getJustUserName() {
        String userName = "";
        /* get the user name */
        return userName;
    }

    @GET
    @Path("/id")
    public String getUserId() {
        String userId = "";
        /* get the user id */
        return userId;
    }
}
```

An HTTP URL request to the **"/administrator"** would resolve to Administrator#findUserInfo(). A HTTP URL request to "/administrator/name" would invoke the Administrator#getJustUserName() method. Finally a HTTP URL request to "/administrator/id" would resolve to Administrator#getUserId().

Using Subresource Locators (@Path on Java method)

In more complicated scenarios, subresource locators are needed. Subresource locators are particularly useful when requests must be further resolved by other objects. Subresource locators are Java methods which have only an **@Path** annotation. They are different than subresource methods because they do **not** have any HTTP method annotation on them.

A subresource locator returns an object that has JAX-RS annotations on its methods (or inherits them). The object is used to further resolve the incoming requests by dynamically inspecting the object for JAX-RS annotations.

This scenario uses @PathParams which are discussed on the parameters page.

```
@Path("/users")
public class UsersCollection {

    @Path("{userid}")
    public Object findUserInfo(@PathParam("userid") String userId) {
        if(userId.equals("superuser")) {
            return new SuperUser();
        }
        return User.findUserInDatabase(userId);
    }
```

```
}

public class Superuser {
    @GET
    public String getUserInfo() {
      String userInfo = /* get the user info */;
      return userInfo;
    }

    @GET
    @Path("/contactinfo")
    public String getContactInfo() {
     String contactInfo = /* get the user contact info */;
     return contactInfo;
    }
}

public class User {
    protected String name;

    protected User() {
        /* subresource locator object lifecycles are controlled by the developer */
    }

    public static User findUserInDatabase(String userName) {
        User u = /* get user from database with assigned field values */
        return u;
    }

    @GET
    public String getInfo() {
        String info = /* get the user info */;
        return info;
    }

    @GET
    @Path("/name")
    public String getMyUserName() {
        return name;
    }
}
```

A HTTP GET to "/users/superuser" would result in User#findUserInfo() being invoked first. The method inspects the "userId" path parameter (which resolves to "superuser" for this request) so a Superuser object is returned. The request is then further resolved against the Superuser object. In the simple "/users/superuser" case, the request invokes Superuser#getUserInfo();

If a HTTP GET to "/users/superuser/contactinfo" was made, then User#findUserInfo() is invoked and again returns a Superuser object. The Superuser object is then used to resolve the "/contactinfo" portion of the request which invokes Superuser#getContactInfo.

If a HTTP GET to "/users/user1/name" was made, then User#findUserInfo() is again invoked but it would go and look up the user from the database and return a User object. The User object is then used to resolve the "/name" portion and results in the User#getMyUserName() method being invoked on the User object returned.

This page last changed on Oct 14, 2009 by michael.

Request and Response Entities

Request and response entities represent the main part of an HTTP request. Entities are also refered to as the "**message body**" or "**payload**". Entities are sent via a request, usually an HTTP POST and PUT method are used or they are returned in a response, this is relevant for all HTTP methods.

Unlike other distributed systems technologies, there is generally no wrapper around an entity. For example, if a request is made for a binary PNG image represented here, http://example.com/user/abcd/portrait.png , the response entity is only the PNG image binary data.

Resource methods have a single entity parameter that represents the main entity body. It is the **only unannotated parameter allowed** in a resource method.

When using JAX-RS, request and response entites are mapped to and from Java types by Entity Providers that implement the JAX-RS interfaces, MessageBodyReader and MessageBodyWriter. Applications may provide their own MessageBodyReaders and MessageBodyWriters that take precedent over the runtime provided ones.

Media Types (MIME) and javax.ws.rs.core.MediaType

The request and response entity can be any form of data, a way of identifying what the entities bits and bytes represent is needed. In requests and responses, the **Content-Type** HTTP header is used to indicate the type of entity currently being sent. The Content-Type value comes from a well known media type as registered in IANA.

Common content types include "text/plain", "text/xml", "text/html", and "application/json".

Correct Content-Type values are essential for clients and servers. "**Unusual**" behavior by clients such as browsers can be attributed to wrong content types.

Media Types are also used in a request **Accept** header to indicate what type of resource representation the client wants to receive. Clients could indicate a preference as well, such as JSON before XML.

> **Reference**
> Refer to the HTTP spec regarding the Accept header and the Content Negotiation topic.

javax.ws.rs.core.MediaType has functionality and representations related to Media Types.

@Consumes and @Produces Annotations

Annotating a class or a resource method with @Consumes and @Produces will help the JAX-RS runtime identify the appropriate methods to invoke for requests. For example:

```
@Path("/example")
public RootResource {
    @POST
    @Consumes("text/xml")
    @Produces("text/xml")
    public Response getOnlyXML(String incomingXML) {
        return Response.ok("only xml").type("text/xml").build();
    }

    @GET
    @Produces("text/html", "text/plain")
    public String getText() {
```

```
        return "text representation";
    }
}
```

In the previous code example, if a HTTP POST to "/example" was issued with a Content-Type header of "text/xml" and an Accept header of "text/xml", then the RootResource#getOnlyXML method would be invoked. If the same POST request was issued with an Accept header of "text/plain", then a 406 Not Acceptable response would be generated by the JAX-RS runtime and the method would not be invoked.

It is a good practice to return a javax.ws.rs.core.Response with a .type() or .variant() call since it would guarantee a return content type. Notice that the above getText() code supports multiple data types. A javax.ws.rs.core.Response object returned must have a single concrete Content-Type value. In orer to select the best acceptable representation in the resource method, use either the **@Context HttpHeaders#getAcceptableMediaTypes()** or a **@Context Request#selectVariant()** method.

Reference
Refer to the Context topic page for more information.

While resource methods may consume one media type for example XML and produce another such as JSON, most user requests expect the same media type that was sent in the request to be returned in the response.

If the Content-Type header is empty and there is an entity, then the JAX-RS runtime will make the Content-Type be "**application/octet-stream**". If an Accept header is empty, then according to the HTTP specification, the Accept header is equivalent to */* which is a wildcard that matches anything.

Important Note
Note that the resource method ultimately has control over the response content. If a javax.ws.rs.core.Response is returned, then the developer can return whatever Content-Type is desired. The @Consumes and @Produces is primarily useful only when processing request information and determining which resource method is possible to invoke. If a specific Response content type is not specified via a returned javax.ws.rs.core.Response object, the response media type is determined by a combination of the @Produces annotation values and the available MessageBodyWriters for the response entity's Java type. This can lead to undesired results if there is no @Produces annotation or if the @Produces annotation has multiple media types listed.

Reference
Refer to the JAX-RS specification for the effective algorithm used.

JAX-RS Standard Entity Parameter Types

JAX-RS requires certain parameters to be supported for virtually any content type. The following table lists the supported content types:

Java Type	Content Type Supported	Special Notes
java.lang.String	*/*	
byte[]	*/*	
java.io.InputStream	*/*	
java.io.Reader	*/*	
java.io.File	*/*	
javax.activation.DataSource	*/*	

javax.xml.transform.Source	text/xml, application/xml, application/*+xml	
javax.xml.bind.JAXBElement and JAXB classes	text/xml, application/xml, application/*+xml	
javax.ws.rs.core .MultivaluedMap <String, String>	application/x-www-form-urlencoded	
javax.ws.rs .core.StreamingOutput	*/*	As a writer only

Developers can use the previous Java types as entity parameters for requests and responses.

```
@Path("/example")
public class RootResource {

    @GET
    @Produces("text/xml")
    public Response getInfo() {
        byte[] entity = /* get the entity into a byte array */
        return Response.ok(entity).type("text/xml").build();
    }

    @POST
    @Consumes("application/json")
    @Produces("application/json")
    public StreamingOutput createItem(InputStream requestBodyStream) {
        /* read the requestBodyStream like a normal input stream */
        return new StreamingOutput() {

            public void write(OutputStream output) throws IOException, WebApplicationException {
                byte[] out = /* get some bytes to write */
                output.write(out);
            }
        })
    }
}
```

Transfer Encoding

Transfer or "**chunked**" encoding is handled by the container for incoming requests. The container or the application must do any transfer encoding for outgoing responses.

Content Encoding

Content for example "**gzip**" and or "**deflate**" encoding is handled by the application. However, some containers handle content encoding for developers and will uncompress content automatically or will with various configuration set. Check the container documentation.

This page last changed on Sep 22, 2009 by michael.

Parameters

Parameters are used to pass and add additional information to a request. Search queries, offsets/pages in a collection, and other information can be used. While parameters are sometimes used to add more verbs to HTTP, it is advised not to use parameters as a way to create new HTTP methods or to make existing HTTP methods break the generally understood attributes (i.e. idempotent).

Parameters can be any basic Java primitive type including java.lang.String as well as types with a constructor that takes in a single String or a valueOf(String) static method.In addition, List, SortedSet, and Set can be used where the generic type is one of the previously mentioned types such as a **Set<String>** when a parameter can have multiple values.

When full control is needed for parsing requests, it is generally recommend that developers use a String as the parameter type so that some basic inspection can be performed and developers can customize error path responses.

Query Parameters (@QueryParam)

Query parameters are one of the better known parameters. A query string is appended to the request URL with a leading "?" and then name/value pairs.

Query Parameter Examples
Refer to the following links:
http://www.google.com/search?q=apache+wink
http://www.example.com/users?offset=100&numPerPage=20

In order to enable JAX-RS to read a query parameters, add a parameter to the resource method and annotate with **@QueryParam**:

```
@Path("/example")
public class RootResource {
    @GET
    public Response invokeWithParameters(@QueryParam("q") String searchTerm) {
        if(q == null) {
            return Response.status(Status.BAD_REQUEST).build();
        }
        /* do a search */
        return Response.ok(/* some entity here */).build();
    }
}
```

If a HTTP GET request to "/example?q=abcd" is made, searchTerm will have "abcd" as the value when invoked.

Path Parameters (@PathParam)

Path parameters take the incoming URL and match parts of the path as a parameter. By including *{name}* in a **@Path** annotation, the resource can later access the matching part of the URI to a path parameter with the corresponding "**name**". Path parameters make parts of the request URL as parameters which can be useful in embedding request parameter information to a simple URL.

```
@Path("/books/{bookid}")
```

```
public class BookResource {
    @GET
    public Response invokeWithBookId(@PathParam("bookid") String bookId) {
        /* get the info for bookId */
        return Response.ok(/* some entity here */).build();
    }

    @GET
    @Path("{language}")
    public Response invokeWithBookIdAndLanguage(@PathParam("bookid") String bookId,
 @PathParam("language") String language) {
        /* get the info for bookId */
        return Response.ok(/* some entity here */).build();
    }
}
```

In the previous example, HTTP GET to /books/1 or to /books/abcd would result in invokeWithBookId being called. If a HTTP GET request is issued to /books/1/en or /books/1/fr or /books/abcd/jp, then invokeWithBookIdAndLanguage would be invoked with the appropriate parameters.

Matrix Parameters (@MatrixParam)

Matrix parameters are not as widely used on the Internet today. However, you can read a MatrixParam just as easily as any other parameter.

```
@Path("/")
public class RootResource {
    @GET
    public Response invokeWithParameters(@MatrixParam("name") String name) {
        if(name == null) {
            return Response.status(Status.BAD_REQUEST).build();
        }
        return Response.ok(/* some entity here */).build();
    }
}
```

Header Parameters (@HeaderParam)

Header parameters are useful especially when there are additional metadata control parameters that need to be passed in for example, security information, cache information, and so forth.

```
@Path("/")
public class RootResource {
    @GET
    public Response invokeWithParameters(@HeaderParam("controlInfo") String controlInfo) {
        if(controlInfo == null) {
            return Response.status(Status.BAD_REQUEST).build();
        }
        return Response.ok(/* some entity here */).build();
    }
}
```

CookieParameters (@CookieParam)

In a REST application, requests are stateless although applications sometimes use Cookies for various reasons, such as adding some stateless resource viewing information without embedding it into the URL such as the maximum number of items to retrieve. The CookieParam annotation is used to easily retrieve the information.

```
@Path("/")
public class RootResource {
   @GET
   public Response invokeWithParameters(@CookieParam("max") String maximumItems) {
      if(userId == null) {
         return Response.status(Status.BAD_REQUEST).build();
      }
      return Response.ok(/* some entity here */).build();
   }
}
```

This page last changed on Sep 22, 2009 by michael.

HTTP Headers

HTTP headers generally store metadata and control information. There are some common headers shared in requests and responses but there are a few specific headers to either a request or a response. Developers should read the HTTP specification for a complete understanding of HTTP headers. Some of the more common HTTP headers are mentioned below in cases where JAX-RS provides convenient methods for the header.

Generally, in order to get the request header name and values, developers can use either an injected @HeaderParam annotated with a parameter/field/property or an injected @Context HttpHeaders parameter/field/property.

```
@Path("/example")
public ExampleResource {
   @Context HttpHeaders requestHeaders;

   @GET
   public void getSomething(@HeaderParam("User-Agent") String theUserAgent) {
      /* use theUserAgent variable or requestHeader's methods to get more info */
   }
}
```

In order to set response headers, developers can set them on a javax.ws.rs.core.Response return object.

```
@Path("/example")
public ExampleResource {
   @GET
   public Response getSomething() {
      return Response.ok(/* some entity */).header("CustomHeader", "CustomValue").build();
   }
}
```

A response headers can be set when a MessageBodyWriter#writeTo() method is called.

Common Headers

The common header specifies the size and type of a header. Every header must begin with the common header. The common header must not appear by itself.

Content-Type

The Content-Type signals the media type of the request/response entity. The Content-Type header on requests is read via HttpHeaders#getMediaType() method. The Content-Type is set for responses when doing a javax.ws.rs.core.Response.ResponseBuilder#type() method or a javax.ws.rs.core.Response.ResponseBuilder#variant() method.

Content-Language

The Content-Language denotes what language the entity is in. In order to receive the request entity language, use the HttpHeaders#getLanguage() method. In order to set the response entity language, use the javax.ws.rs.core.Response.ResponseBuilder#language() method.

Content-Length

The Content-Length is useful for determining the entity's length. If possible, the MessageBodyWriter entity providers will automatically set the Content-Length if possible, and some containers will set the response Content-Length if the entity is small.

> **Reference**
> Refer to the HTTP spec for more details on when this header is set and valid.

Common Request HTTP Headers

An HTTP Request Header is a line of text that a client software (i.e. Web Browser) sends to a server together with other request data.

Accept

The Accept header is used to determine the possible response representations that the client prefers such as XML over JSON but not plain text. When a resource method is effectively annotated with a @Produces, any incoming request must have a compatible Accept header value for the resource method to be selected. Clients can set quality parameters (priority ordering) for the best possible response and services generally try to honor the request. To get the best representation of a response, use either the HttpHeaders#getAcceptableMediaTypes() or Request#selectVariant() methods.

Accept-Language

Like the Accept header, Accept-Language lists the preferred languages. A HttpHeaders#getAcceptableLanguages() method will list the acceptable languages in a preferred order.

If-Match and If-None-Match

If a previous response had an ETag header, the client can re-use the ETag value with the If-Match or If-None-Match request header to do conditional requests (if the server application supported the If-Match/If-None-Match headers). For example, a POST with an If-Match header and an old ETag value should only execute the POST request if the old ETag value is still valid. javax.ws.rs.core.Request#evaluatePreconditions() may help evaluate the If-Match and If-None-Match headers.

If-Modified-Since and If-Unmodified-Since

Like ETags, If-Modified-Since and If-Unmodified-Since are based on the Last-Modified date. Using either request header with a date, will set up a conditional request (if the server application supports the headers). For instance, a GET with an If-Modified-Since header of an old known date would allow the server to send back a response with just a HTTP status code of 304 (Not Modified). By sending back a HTTP status code of 304, the server is telling the client that the resource has not changed so there is no need to re-download the resource representation. This could save precious bandwidth for the client. javax.ws.rs.core.Request#evaluatePreconditions() may help evaluate the If-Modified-Since and If-Unmodified-Since headers.

> **Important Note**
> Note that date formats are specified in the HTTP specification.

Common Response HTTP Headers

HTTP Headers form the core of an HTTP request, and are very important in an HTTP response. They define various characteristics of the data that is requested or the data that has been provided. The headers are separated from the request or response body by a blank line

ETag

ETags or entity tags can be set. Like a hashcode, it is given to the client so a client can use various control request headers such as If-Match and If-None-Match for conditional requests. javax.ws.rs.core.Response.ResponseBuilder#tag() and javax.ws.rs.core.EntityTag are useful for ETags.

Expires

The Expires response header indicates the amount of time that the response entity should be cached. It is useful to set the expiration for data that is not going to change for a known time length. Browsers use this response header to manage their caches among other user agents.The javax.ws.rs.core.Response.ResponseBuilder#expires() method can be used to set the Expires header.

Last-Modified

Last-Modified specifies the date when the resource was changed. A client can use the response value in combination with If-Modified-Since and If-Unmodified-Since request headers to perform conditional requests.The javax.ws.rs.core.Response.ResponseBuilder#lastModified() method can be used to set the Last-Modified header.

Important Note
Note that date formats are specified in the HTTP specification.

Location

The Location response header usually indicates where the resource is located (in a redirect) or the URI of the new resource (when resources are created and usually in a HTTP 201 response). The javax.ws.rs.core.Response.ResponseBuilder#location()method can be used to set the Location header.

JAX-RS Content Negotiation

This page last changed on Sep 22, 2009 by michael.

What is Content Negotiation?

One of the more popular features of REST applications is the ability to return different representations of resources. For instance, data can be in JSON format or in XML. Or it can even be available in either format depending on the request. Content negotiation is the way for clients and servers to agree on what content format is sent.

Data is returned in multiple formats because the needs of each client's request can be different. A browser might prefer JSON format. Another command line based client might prefer XML format. A third client might request the same resource as a PNG image.

It is up to the service to determine which formats are supported.

There are many practical ways of performing content negotiation.

Content Negotiation Based on URL

Many popular public REST APIs perform content negotiation based on the URL. For instance, a resource in XML format might be at http://example.com/resource.xml. The same resource could be offered in JSON format but located at http://example.com/resource.json.

```
@Path("/resources")
public class Resource {

   @Path("{resourceID}.xml")
   @GET
   public Response getResourceInXML(@PathParam("resourceID") String resourceID) {
      return Response.ok(/* entity in XML format */).type(MediaType.APPLICATION_XML).build();
   }

   @Path("{resourceID}.json")
   @GET
   public Response getResourceInJSON(@PathParam("resourceID") String resourceID) {
      return Response.ok(/* entity in JSON format */).type(MediaType.APPLICATION_JSON).build();
   }
}
```

In the above code, a request to **"/resources/resourceID.myformat"** would result in a 404 status code.

Another way of implementing the above is to use a single resource method like below:

```
@Path("/resources")
public class Resource {

   @Path("{resourceID}.{type}")
   @GET
   public Response getResource(@PathParam("resourceID") String resourceID, @PathParam("type") String
 type) {
      if ("xml".equals(type)) {
         return Response.ok(/* entity in XML format */).type(MediaType.APPLICATION_XML).build();
      } else if ("json".equals(type)) {
         return Response.ok("/* entity in JSON format */).type(MediaType.APPLICATION_JSON).build();
      }
```

```
        return Response.status(404).build();
    }

}
```

The previous code excerpt essentially behaves the same as the ContentNegotiationViaURL.java example.

Content Negotiation Based on Request Parameter

Another popular method is for the client to specify the format in a parameter. For instance, by default, a resource could be offered in XML at http://example.com/resource. The JSON version could be retrieved by using a query parameter like http://example.com/resource?format=json.

```
@Path("/resources")
public class Resource {

    @Path("{resourceID}")
    @GET
    public Response getResource(@PathParam("resourceID") String resourceID,
                                @QueryParam("format") String format) {
        if (format == null || "xml".equals(format)) {
            return Response.ok(/* entity in XML format */).type(MediaType.APPLICATION_XML).build();
        } else if ("json".equals(format)) {
            return Response.ok(/* entity in JSON format */).type(MediaType.APPLICATION_JSON).build();
        }

        return Response.notAcceptable(Variant.mediaTypes(MediaType.APPLICATION_XML_TYPE,
                                                         MediaType.APPLICATION_JSON_TYPE).add()
            .build()).build();
    }
}
```

Content Negotiation Based on Accept Header

Perhaps the most powerful form of content negotiation, the HTTP Accept header is another way of specifying the preferred representation format.

The following excerpt shows sample client code using the Wink RestClient:

```
    RestClient client = new RestClient();
    ClientResponse response = client.resource("http://example.com/resources/
resource1").header("Accept", "application/json;q=1.0, application/xml;q=0.8").get();
    // find what format was sent back
    String contentType = response.getHeaders().getFirst("Content-Type");

    if (contentType.contains("application/json")) {
        JSONObject entity = response.getEntity(JSONObject.class); /* or use a String, InputStream, or other
 provider that supports the entity media type */
    } else if (contentType.contains("application/xml") {
        String entity = response.getEntity(String.class); /* or use a JAXB class, InputStream, etc. */
    }
```

The following example implements sample client code using the Apache HttpClient.

```
HttpClient client = new HttpClient();
GetMethod getMethod =
    new GetMethod("http://example.com/resources/resource1");
// prefer JSON over XML but both are acceptable to the client
getMethod.setRequestHeader("Accept", "application/json;q=1.0, application/xml;q=0.8");
try {
    client.executeMethod(getMethod);

    // find what format was sent back
    getMethod.getResponseHeader("Content-Type").getValue();

    // use getMethod.getResponseBodyAsString() or getMethod.getResponseBodyAsStream()
    // to read the body
} finally {
    getMethod.releaseConnection();
}
```

Using the @Context HttpHeaders injected variable, the client preferred response representation is found.

```
@Path("/resources")
public class Resource {

    @Context
    private HttpHeaders headers;

    @Path("{resourceID}")
    @GET
    public Response getResource(@PathParam("resourceID") String resourceID) {
        List<MediaType> acceptHeaders = headers.getAcceptableMediaTypes();
        if (acceptHeaders == null || acceptHeaders.size() == 0) {
            return Response.ok(/* entity in XML format */).type(MediaType.APPLICATION_XML).build();
        }

        for (MediaType mt : acceptHeaders) {
            String qValue = mt.getParameters().get("q");
            if(qValue != null && Double.valueOf(qValue).doubleValue() == 0.0) {
                break;
            }
            if(MediaType.APPLICATION_XML_TYPE.isCompatible(mt)) {
                return Response.ok(/* entity in XML format */).type(MediaType.APPLICATION_XML).build();
            } else if (MediaType.APPLICATION_JSON_TYPE.isCompatible(mt)) {
                return Response.ok(/* entity in JSON format */).type(MediaType.APPLICATION_JSON).build();
            }
        }
        return Response.notAcceptable(Variant.mediaTypes(MediaType.APPLICATION_JSON_TYPE,
                                            MediaType.APPLICATION_XML_TYPE).add()
            .build()).build();
    }

}
```

If the client request does not have an Accept HTTP header, then by default the XML format is returned. The @Context HttpHeaders.getAcceptableMediaTypes() method returns an ordered list, sorted by the client preference of the representations.

Looping through the acceptable media types, if the preferred media type is compatible with one of the service's available return types, then the preferred media type is used.

Note
Note that the quality factor is checked. In fairly rare requests, clients can let the service know that a media type should not be sent back (if the quality factor is 0.0).

This page last changed on Aug 24, 2009 by bluk.

Context Information

In addition to request parameters and entities, there is more request information that can be accessed via request @Context injected variables.

When a resource method is invoked, the runtime injects the @Context variables.

```
@Path("/resource")
public class Resource {
   @Context
   private HttpHeaders headers;

   @GET
   public void get(@Context UriInfo uriInfo) {
     /* use headers or uriInfo variable here */
   }
}
```

javax.ws.rs.core.HttpHeaders

HttpHeaders provides methods that allow users to access request headers. A few convenience methods such as #getAcceptableLanguages() and #getAcceptableMediaTypes() provide client preference sorted acceptable responses.

javax.ws.rs.core.UriInfo

UriInfo provides methods so developers can find or build URI information of the current request.

javax.ws.rs.core.SecurityContext

SecurityContext provides access to the security information.

javax.ws.rs.core.Request

Request provides methods for evaluating preconditions and for selecting the best response variant based on the request headers.

javax.ws.rs.core.Providers

Providers allows access to the user and runtime provided MessageBodyReaders, MessageBodyWriters, ContextResolvers, and ExceptionMappers. It is useful for other providers but can sometimes be useful for resource methods and classes.

5 Apache Wink Server

This page last changed on Apr 20, 2010 by bluk.

Apache Wink Server Module

The following section describes the Apache Wink Server and provides a detailed description of the Apache Wink Server component and its functionality.

Contents

Apache Wink Server Overview

The Apache Wink Server module is an implementation of the JAX-RS specification. In addition to the core implementation, the Wink Server module provides a set of additional features that are designed to facilitate the development of RESTful Web services. The framework is easy to extend and to enrich with new functionality.

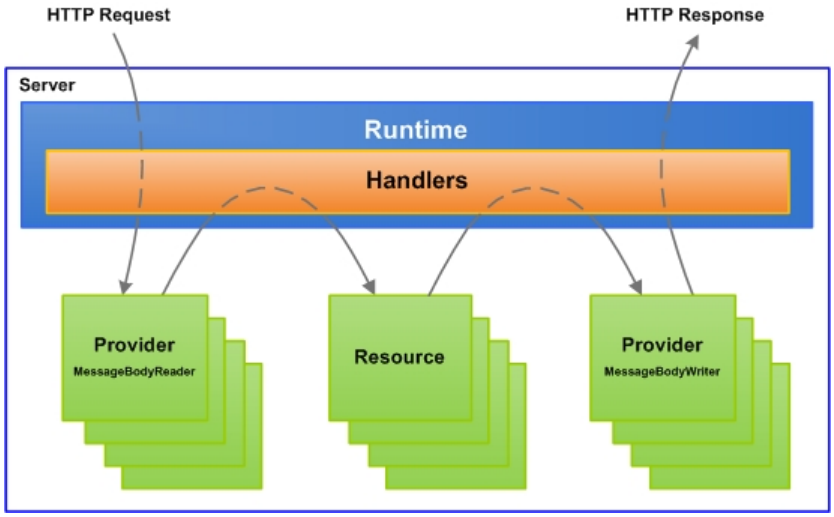
Main Features

The Apache Wink Server main features are as follows:

- Aims to be an implementation of the JAX-RS v1.1 specification
- Provides out-of-the-box Java object models for Atom, Json, RSS, APP, CSV, HTML, Multipart and OpenSearch along with providers to serialize and deserialize these models
- Highly configurable and flexible runtime functionality
- Provides a Handlers mechanism for manipulation of HTTP request and response messages
- Automatic generation of APP document for collection resources
- Spring integration
- Provides support for WebDAV through the WebDAV extension
- Provides an in-depth administration view of the runtime registries

Apache Wink High Level Server Architecture Overview

The following diagram illustrates the general architecture of the Apache Wink server runtime.



The Apache Wink server runtime layer receives incoming HTTP requests from the hosting container. Once a request is received the Apache Wink server runtime initiates a new request session by creating a message context that is passed through the handlers chain which consists of system and user defined handlers.

Initially the runtime passes the message context through the handlers responsible for finding the resources and resource methods that match the request according to the JAX-RS specification. If required, the incoming request is de-serialized using the appropriate provider. Once the injectable parameters are ready for injection the matched resource method is invoked and the returned response object is then passed through the handler chain in order to select and activate the appropriate provider for serialization as the HTTP response.

This page last changed on Oct 13, 2009 by bluk.

Registration and Configuration

Apache Wink provides several methods for registering resources and providers. This chapter describes registration methods and Wink configuration options.

Simple Application

Apache Wink provides the "**SimpleWinkApplication**" class in order to support the loading of resources and providers through a simple text file that contains a list of fully qualified class names of the resource and provider classes. Each line contains a single fully qualified class name that is either a resource or a provider. Empty lines and lines that begin with a number sign (#) are permitted and ignored.

```
# Providers
com.example.MyXmlProvider
com.example.MyJSONProvider

# Resources
com.example.FooResource
com.example.BarResource
```

Specifying the Simple Application File Location

The path to a simple application file is configured via the applicationConfigLocation init-param in the web.xml file. It is possible to specify multiple files by separating them with a semicolon.

```
<servlet>
  <servlet-name>restSdkService</servlet-name>
  <servlet-class>
    org.apache.wink.server.internal.servlet.RestServlet
  </servlet-class>
  <init-param>
    <param-name>applicationConfigLocation</param-name>
    <param-value>/WEB-INF/providers;/WEB-INF/resources</param-value>
  </init-param>
</servlet>
```

Apache Wink Application

Apache Wink extends the javax.ws.rs.core.Application class with the org.apache.wink.common.WinkApplication class in order to provide the Dynamic Resources and the Priorities functionality.

An application may provide an instance of the Apache Wink Application to the Apache Wink runtime as specified by the JAX-RS specification.

Dynamic Resources

Dynamic Resources enable the binding of a Resource class to a URI path during runtime instead of by using the @Path annotation. A dynamic resource must implement the DynamicResource interface and must not be annotated with the @Path annotation.

Motivation

A Dynamic Resource is useful for situations where a resource class must be bound to multiple paths, for example, a sorting resource:

```
public class SortingResource<E extends Comparable<? super E>> {
    private List<E> list;
    @POST
    public void sort() {
        Collections.sort(list);
    }
    public void setList(List<E> list) {
        this.list = list;
    }
    public List<E> getList() {
        return list;
    }
}
```

Explanation

In this example, the SortingResource class can sort any list. If the application manages a library of books and exposes the following resource paths, then the SortingResource class can be used for the implementation of all these resource paths, assuming that it could be bound to more than one path.

```
/sort-books
/sort-authors
/sort-titles
```

A dynamic resource is also useful for situations where the resource path is unknown during development, and is only known during the application startup.

Usage

A Dynamic Resource is a resource class that implements the org.apache.wink.server.DynamicResource interface or extends the org.apache.wink.server.AbstractDynamicResource convenience class.

A Dynamic Resource is not registered in Apache Wink through the Application#getClasses() method or the Application#getSignletons() method, since the same class can be used for multiple resources.
In order to register Dynamic Resources in the system, the WinkApplication#getInstances()method must be used.

Scope

The scope of a Dynamic Resource is limited to "singleton" as it is initialized prior to its registration, and the system does not have enough information to create it in runtime. This limitation is irrelevant when working with Spring. Refer to chapter #0 for more information on Spring integration.

Priorities

Although JAX-RS defines the algorithm for searching for resources and providers, Apache Wink enables to extend this algorithm by allowing the specification of priorities for them.
Apache Wink extends the JAX-RS search algorithms by providing the ability to specify priorities on the resources and providers. This is achieved by enabling the registration of multiple Application instances with different priorities, rendering the order of their registration irrelevant as long as they have different priorities.

In order to register a prioritized Application, it is necessary to register an instance of a Apache WinkApplication class. Priority values range between 0 and 1. In the event that the priority was not specified, a default priority of 0.5 is used.

Resource Priorities

Priorities on resources are useful for situations where an application registers core resources bound to paths, and allows extensions to register resources on the same paths in order to override the core resources.

The Apache Wink runtime first sorts the resources based on their priority and then based on the JAX-RS specification, thus if two resources have the same path, the one with higher priority is invoked.

Provider Priorities

JAX-RS requires that application-provided providers be used in preference to implementation pre-packaged providers. Apache Wink extends this requirement by allowing applications to specify a priority for providers.

The Apache Wink runtime initially sorts the matching providers according to the JAX-RS specification, and uses the priority as the last sorting key for providers of equal standing.
If two providers have the same priority, the order in which they are registered determines their priority such that the latest addition receives the highest priority.
In order to meet the JAX-RS requirements, the pre-packages providers are registered using a priority of 0.1.

Properties Table

Property Name	Description	Default Value
wink.http.uri	URI that is used by the Link Builders in case of HTTP	Use the URI from the request
wink.https.uri	URI used by the Link Builders in case of HTTPS	Use the URI from the request
wink.context.uri	Context path used by the Link Builders	Use the context path from the request
wink.defaultUrisRelative	Indicates if URIs generated by the Link Builders are absolute or relative, valid values: true or false	true - links are relative
wink.addAltParam	Indicates if the "alt" query parameter should be added to URIs generated by the Link Builders. Valid values are: true, false	true - add the alt query parameter
wink.search PolicyContinuedSearch	Indicates if continues search is enabled. Valid values: true, false	true - continued search is enabled
wink.rootResource	Indicates if a root resource with Service Document generation capabilities should be added. Valid values are: none, atom, atom+html	atom+html --atom and html Service Document generation capabilities
wink.serviceDocumentCssPath	Defines path to a css file that is used in the html Service	No css file defined

	Document generation. Relevant only if html Service Document is defined	
wink.handlersFactoryClass	Defines a org.apache.wink.server .handlers.HandlersFactory class that defines user handlers	No user handlers defined
wink.mediaType MapperFactoryClass	Defines a org.apache.wink.server.handlers .MediaTypeMapperFactory class that defines media type mappers	No media type mappers defined
wink.loadApplications	Loads providers defined in a wink-application file found by the runtime	True, automatically load all wink-application specified classes

Custom Properties File Definition

In order to provide a custom properties file, the application should define the propertiesLocation init-param in the Apache Wink Servlet definition.

```
# Providers
<servlet>
  <servlet-name>restSdkService</servlet-name>
  <servlet-class>
    org.apache.wink.server.internal.servlet.RestServlet
  </servlet-class>
  <init-param>
    <param-name>propertiesLocation</param-name>
    <param-value>/WEB-INF/configuration.properties</param-value>
  </init-param>
  <init-param>
    <param-name>winkApplicationConfigLocation</param-name>
    <param-value>/WEB-INF/application</param-value>
  </init-param>
  <load-on-startup>0</load-on-startup>
</servlet>
```

Runtime

RegistrationApache Wink provides several APIs for Runtime Registration. The APIs appear in the org.apache.wink.server.utils.RegistrationUtils class. The most important method is the one that registers an instance of the javax.ws.rs.core.Application class

```
# Providers
static void registerApplication(Application application, ServletContext servletContext)
```

Double Registration
Registration is ignored and a warning is printed to the log if the instance was previously registered

Media-Type Mapping

It is sometimes necessary to override the Content-Type response header based on the client user agent. For example, the Firefox browser cannot handle the application/atom+xml media type for Atom content, unless it is defined as a text/xml.

Apache Wink provides a set of predefined Media-Type mappings for use in such cases by supplying the MediaTypeMapper class. Applications may extend or override the MediaTypeMapper class to define additional mappings.

Customizing Mappings

In order to customize these mappings the application should create a instance of a org.apache.wink.server.handlers.MediaTypeMapperFactory class and set it in a customized Wink properties file.

> **Reference**
> Refer to section 5.1 Registration and Configuration for more information on Customizing the Default Properties Configuration.

Alternative Shortcuts

Clients specify the requested media type by setting the Http Accept header. Apache Wink provides an alternate method for specifying the requested media type via use of the "alt" request parameter. This functionality is useful for situation where the client has little affect on the accept header, for example when requesting a resource using a browser.

For example, a request to /entry?alt=application/xml specifies that the requested response media type is application/xml. Apache Wink provides a shortcut mechanism for specifying the media type of the alt query parameter and provides a predefined set of shortcuts for common media types.

Predefined Shortcuts

Shortcut	Media Type
json	text/javascript
atom	application/atom+xml
xml	application/xml
text	text/plain
html	text/html
csv	text/csv
opensearch	application/opensearchdescription+xml

Customizing Shortcuts

The shortcuts table can be customized by overriding the deployment configuration class.

> **Reference**
> Refer to section 2 Apache Wink Building Blocks for more information on Customizing the Default Deployment Configuration.

5.2 Annotations

This page last changed on Oct 13, 2009 by michael.

Annotations

Apache Wink provides several annotations in addition to those defined by the JAX-RS specification. The following section describes these annotations in detail.

@Workspace Annotation

The purpose of the @Workspace annotation is to associate a "Collection Resource" with a workspace element and collection elements in an APP Service Document.

> **Reference**
> For more information regarding the APP Service Document, refer to section 5.4 APP Service Document

The workspaceTitle annotation parameter specifies the title of the workspace and the collectionTitle annotation parameter specifies the title of the collection.

@Workspace Annotation Specification

Value	Description	
Mandatory	No	
Target	Resource class	
Parameters	**Name**	**Type**
	workspaceTitle	String
	collectionTitle	String
Example	@Workspace(workspaceTitle = "Title", collectionTitle = "Collection") x	

@Workspace Annotation Example

The following example demonstrates the use of @Workspace annotation on two resources in order to have the auto-generated APP service document contain the information about them.

Given the following collection Resources definitions, ResourceA and ResourceB, the result is displayed in the "Auto Generated APP Service Document" table that follows.

ResourceA Definition

```
@Workspace(workspaceTitle = "Services", collectionTitle = "Service1")
@Path("services/service1")
public class ResourceA {
    @POST
    @Produces("text/plain")
    @Consumes({"application/atom+xml", "application/xml"})
    public String getText() {return "hey there1";}
```

```
}
```

ResourceB Definition

```
@Workspace(workspaceTitle = "Services", collectionTitle = "Service2")
@Path("services/service2")
public class ResourceB {
    @POST
    @Produces("text/plain")
    @Consumes({"application/atom+xml", "application/xml"})
    public String getText() {return "hey there2";}
}
```

The auto-generated APP Service Document is as follows:

Auto Generated APP Service Document

```
<service xmlns:atom=http://www.w3.org/2005/Atom
        xmlns="http://www.w3.org/2007/app">
    <workspace>
        <atom:title>Services</atom:title>
        <collection href="services/service1">
            <atom:title>Service1</atom:title>
            <accept>application/xml</accept>
            <accept>application/atom+xml</accept>
        </collection>
        <collection href="services/service2">
            <atom:title>Service2</atom:title>
            <accept>application/xml</accept>
            <accept>application/atom+xml</accept>
        </collection>
    </workspace>
</service>
```

@Asset Annotation

The @Asset annotation is a marker annotation used by the Apache Wink runtime in order to identify an entity as an Asset.

> **Reference**
> For more information about Assets refer to section 5.9 Assets.

@Asset Annotation Specification

Value	Description
Mandatory	No
Target	Resource class
Parameters	None
Example	@Asset

@Scope Annotation

The JAX-RS specification defines the default lifecycle behavior for resources and providers, and the option for controlling the lifecycle through the javax.ws.rs.core.Application class.

Apache Wink provides the @Scope annotation to specify the lifecycle of a provider or resource.

@Scope Annotation Specification

Value	Description	
Mandatory	No	
Target	Provider class or Resource class	
Parameters	**Name**	**Type**
	Value	ScopeType enum
Example	@Scope(ScopeType.PROTOTYPE)	

Resource Example

The following example illustrates how to define a resource with a singleton lifecycle.

```
@Scope(ScopeType.SINGLETON)
@Path("service1")
public class ResourceA {
   ...
}
```

Provider Example

The following example illustrates how to define a provider with a prototype lifecycle.

```
@Scope(ScopeType.PROTOTYPE)
@Provider
public class EntityProvider implements MessageBodyReader<String> {
   ...
}
```

@Parent Annotation

The @Parent annotation provides the ability to define a base template URI for the URI specified in a resources @Path annotation.
If a resource is annotated with the @Parent annotation, the Apache Wink runtime calculates the final resource template by first retrieving the value of the @Parent annotation, which holds the parent resource class, and then concatenates the resource path template definition to the path template definition of the parent resource.

@Parent Annotation Specification

Value	Description	
Mandatory	No	
Target	Provider class or Resource class	
Parameters	**Name**	**Type**
	Value	Class<?>
Example	@Parent(ParentResource.class)	

Example 1

```
@Path("services")
public class ParentResource {
   ...
}
```

Example 2

```
@Parent(BaseResource.class)
@Path("service1")
public class ResourceA {
   ...
}
```

Explanation

In the example, the user defined two resources: A ParentResource and ResourceA. ParentResource defines the @Path annotation to associate it with "**services**" URI. ResourceA defines the @Path annotation to associate it with "**service1**" URI and defines ParentResource to be its parent by specifying it in the @Parent annotation. In this case, the final URI path for ResourceA is "**services/service1**".

5.3 Resource Matching

This page last changed on Oct 13, 2009 by michael.

Resource Matching

Apache Wink provides a Continued Search mode when searching for a resource method to invoke during request processing, which is an extended search mode to the algorithm defined by the JAX-RS specification.

Resource Matching Overview

Section 3.7.2 of the JAX-RS specification describes the process of matching requests to resource methods. The fact that only the first matching root resource (section 1(f) of the algorithm) and only the first matching sub-resource locator (section 2(g) of the algorithm) are selected during the process makes it difficult for application developers to implement certain scenarios.
For example, it is impossible to have two resources anchored to the same URI, each having its own set of supported methods:

```
@Path("my/service")
public class ResourceA {
    @GET
    @Produces("text/plain")
    public String getText() {...}
}

@Path("my/service")
public class ResourceB {
    @GET
    @Produces("text/html")
    public String getHtml() {...}
}
```

Explanation

In order to implement this according to the JAX-RS specification, ResourceB must extend ResourceA and be registered instead of ResourceA. However, this may not always be possible, such as in an application that uses JAX-RS as the web service frontend while providing an open architecture for registering extending services. For example, Firefox that provides an Extensions mechanism. The extending service must be aware of the core implementation workings and classes, that may not always be plausible. Moreover, it is impossible for a service to extend the functionality of another service without knowing the inner workings of that service, that creates an "evil" dependency between service implementations.

In order to solve this problem, Apache Wink provides a special resource Continued Search mode when searching for a resource and method to invoke. By default, this mode is off, meaning that the search algorithm is strictly JAX-RS compliant. When this mode is activated, and a root resource or sub-resource locator proves to be a dead-end, the Apache Wink runtime will continue to search from the next root-resource or sub-resource locator, as if they were the first match.

In the previous example, there is no way to know which of the resources is a first match for a request to "**/my/service**". If the Continued Search mode is off, either the getText() method is unreachable or the getHtml() method is unreachable. However, when the Continued Search mode is active, a request for text/plain reaches the getText() method in ResourceA, and a request for text/html reaches the getHtml() method in ResourceB.

Configuration

The Continued Search mode is activated by setting the value of the wink.searchPolicyContinuedSearch key in the application configuration properties file to true.
If the key is set to anything else but true or if it does not exist in the properties file, then the Continued Search mode is set to off, and the behavior is strictly JAX-RS compliant.

This page last changed on Oct 13, 2009 by michael.

APP Service Document

Apache Wink supports the automatic and manual generation of APP Service Documents by providing an APP data model and set of complementary providers.

Atom Publishing Protocol Service Documents are designed to support the auto-discovery of services. APP Service Documents represent server-defined groups of Collections used to initialize the process of creating and editing resources. These groups of collections are called Workspaces. The Service Document can indicate which media types and categories a collection accepts.

The Apache Wink runtime supports the generation of the APP Service Documents in the XML (application/atomsvc+xml) and HTML (text/html) representations.

Enabling the APP Service Document Auto Generation

APP Service Document generation is activated by setting the wink.rootResource key in the configuration properties file. By default, the key value is set to "atom+html", indicating that both XML (application/atomsvc+xml) and HTML (text/html) representations are available.

Once activated, the auto-generated APP Service Document is available at the application root URL "http://host:port/application".

Adding Resources to APP Service Document

Apache Wink provides the @Workspace annotation used to associate a Collection Resource with an APP Service Document workspace and collection elements. The only requirement to incorporate a collection resource in a service document is to place the @Workspace annotation on the resource.

Reference
For more information on the @Workspace annotation refer to 5.2 Annotations.

Example

Given the following collection resource definition:

```
@Workspace(workspaceTitle = "Workspace", collectionTitle = "Title")
@Path("my/service")
public class ResourceA {
   ...
}
```

The auto-generated APP Service Document is:

```
<service xmlns:atom=http://www.w3.org/2005/Atom
       xmlns="http://www.w3.org/2007/app">
  <workspace>
    <atom:title>Workspace</atom:title>
    <collection href="my/service">
      <atom:title>Title</atom:title>
      <accept/>
    </collection>
  </workspace>
```

```
</service>
```

APP Service Document HTML Styling

Apache Wink provides the ability to change the default styling of the APP Service Document HTML representation. The styling is changed by setting the value of the wink.serviceDocumentCssPath key in the configuration properties file to the application specific CSS file location.

Implementation

The following classes implement the APP Service Document support:

- **org.apache.wink.server.internal.resources.RootResource** - generates the XML (application/atomsvc+xml) representation of the APP Service Document.

org.apache.wink.server.internal.resources. HtmlServiceDocumentResource - generates the HTML (text/html) representation of the APP Service Document.

5.5 Spring Integration

This page last changed on Oct 12, 2009 by bluk.

Spring Integration

Apache Wink provides an additional module deployed as an external jar in order to provide Spring integration. The Spring integration provides the following features:

- The ability to register resources and providers from the Spring context, registered as classes or as Spring beans
- The ability to define the lifecycle of resources or providers that are registered as Spring beans, overriding the default scope specified by the JAX-RS specification.
- Resources and providers can benefit from Spring features such as IoC and post-processors.
- Customize Apache Wink from the Spring context. When working with Spring, Apache Wink defines a core spring context that contains customization hooks, enabling easy customization that would otherwise require coding.

Spring Registration

Spring makes it convenient to register resources and providers as spring beans.
Spring Context Loading. In order to load the Spring Context, it is necessary to add a Context Load Listener definition to the web.xml file. The contextConfigLocation context-param must specify the location of the Apache Wink core context file and the application context file, as described in the following example:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:META-INF/server/winkCoreContext-server.xml
            classpath:mycontext.xml
  </param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

Registering Resources and Providers

Apache Wink provides the org.apache.wink.spring.Registrar class in order to register resources and providers through a Spring context. The Registrar class extends the WinkApplication class and must be registered as a singleton spring bean. It is possible to define multiple registrars in the same context. All registrars are automatically collected by the runtime and registered as WinkApplication objects during the context loading. The registrar provides the following properties:

- **instances** - instances of resources and providers. Ordinarily, these instances are Spring beans, so they can benefit from IoC and other Spring features.
- **classes** - a set of resources and providers class names. This property is similar to the getClasses() method of the Application class.
- **priority** - the priority of the WinkApplication

Reference

For more information on Priorities refer to section 5.1 Registration and Configuration.

```
<bean class="org.apache.wink.spring.Registrar">
  <property name="classes">
    <set value-type="java.lang.Class">
      <value>package.className</value>
    </set>
  </property>
  <property name="instances">
    <set>
      <ref bean="resources.resource1" />
      <ref bean="resources.resource2" />
      <ref bean="providers.provider1" />
    </set>
  </property>
</bean>
```

Custom Properties File Definition

Apache Wink provides a set of customizable properties. When working with Spring, the user should redefine the custom properties file using the Spring context:

```
<bean id="customPropertiesFactory"
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="locations">
    <list>
      <value>WEB-INF/configuration.properties</value>
    </list>
  </property>
</bean>
<bean id="customConfigurer"
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="ignoreUnresolvablePlaceholders" value="true" />
  <property name="order" value="1" />
  <property name="propertiesArray">
    <list>
      <props>
       <prop key="wink.propertiesFactory">customPropertiesFactory</prop>
      </props>
    </list>
  </property>
</bean>
```

- The customPropertiesFactory bean loads the properties file.
- The customConfigurer bean overrides the default factory with a custom factory.
- The order is set to "1". This makes the customConfigurer bean run before the default Apache Wink configurer.
- In addition, notice that ignoreUnresolvablePlaceholders must be set to true, otherwise the configurer will fail, since some unresolved properties can remain in the context.

Customizing Media-Type Mappings

Apache Wink provides the ability to customize the Media-Type mappings using Spring context.

Reference
For more information on Media-Type Mapping refer to section 5.1 Registration and Configuration .

```
<bean id="custom.MediaTypeMapper" class="org.apache.wink.server.internal.MediaTypeMapper">
  <property name="mappings">
    <list>
      <map>
        <entry key="userAgentStartsWith" value="Mozilla/" />
        <entry key="resultMediaType">
          <util:constant static-field=" javax.ws.rs.core.MediaType.ATOM" />
        </entry>
        <entry key="typeToSend">
          <util:constant static-field="javax.ws.rs.core.MediaType.TEXT_XML" />
        </entry>
      </map>
    </list>
  </property>
</bean>
<bean id="customConfigurer"
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="ignoreUnresolvablePlaceholders" value="true" />
  <property name="order" value="1" />
  <property name="propertiesArray">
    <list>
      <props>
        <prop key="wink.MediaTypeMapper">custom.MediaTypeMapper</prop>
      </props>
    </list>
  </property>
</bean>
```

- The custom.MediaTypeMapper bean creates a new Media-Type mapper.
- The customConfigurer bean overrides the default factory with a custom factory.

customConfigurer
The order is set to "1". This makes the customConfigurer run before the default Apache Wink configurer.

* In addition, notice that ignoreUnresolvablePlaceholders must be set to true, otherwise the configurer will fail, since some unresolved properties can remain in the context.

Customizing Alternative Shortcuts

Apache Wink provides the ability to customize the Alternative Shortcuts in one of two ways.

Reference
For more information on Alternative Shortcuts Mappings refer to section 5.1 Registration and Configuration.

External Properties File

The shortcuts are defined in a properties file. The shortcuts properties file is loaded in the same way that the configuration properties file is loaded.

```
<bean id="custom.Shortcuts"
```

```
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="locations">
    <list>
      <value>WEB-INF/shortcuts</value>
    </list>
  </property>
</bean>
<bean id="customConfigurer"
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="ignoreUnresolvablePlaceholders" value="true" />
  <property name="order" value="1" />
  <property name="propertiesArray">
    <list>
      <props>
        <prop key="wink.alternateShortcutsMap">custom.Shortcuts</prop>
      </props>
    </list>
  </property>
</bean>
```

Spring Context File

Defines the map of the shortcuts in the Spring context.

This page last changed on Oct 13, 2009 by michael.

WebDAV Extension

Apache Wink provides an extension module for supporting the WebDAV protocol. The extension contains the complete WebDAV XML model and a WebDAV response builder for easing the process of creating a WebDAV multistatus response.

> **Usful Information**
> The WebDAV extension is a single jar wink-webdav-<version>.jar, and it has no special dependencies.

WebDAV Data Model

The WebDAV extension provides a Java data model that reflects the WebDAV XMLs defined in the WebDAV RFC. All classes of the data model are located in the org.apache.wink.webdav.model package.

WebDAV Classes

The WebDAV extension provides several classes that applications can use in order to receive basic support for common WebDAV methods.

WebDAVModelHelper

The WebDAVModelHelper class provides helper methods for XML marshaling and unmarshaling of the WebDAV data model classes. It also provides helper methods for creating generic properties as DOM element classes to populate the WebDAV Prop element.

WebDAVResponseBuilder

The WebDAVResponseBuilder class is used in order to create responses to WebDAV PROPFIND requests. It takes a SyndEntry or SyndFeed as input in order to create the response.

Resource Method Definition

A resource method is defined to handle the desired WebDAV method by annotating it with one of the WebDAV method designators defined in the WebDAVMethod enum.
The supported WebDAV Http methods are as follows:

- PROPFIND
- PROPPATCH
- MKCOL
- COPY
- MOVE
- LOCK
- UNLOCK.

Creating a Multistatus Response

In order to create a **MULTISTATUS** response to a **PROPFIND** request the user can use the **WebDAVResponseBuilder** class, or create the response manually.

Using WebDAVResponseBuilder

In order to create a multistatus response using the **WebDAVResponseBuilder** class, call one of the propfind() methods.

The **WebDAVResponseBuilder** class also enables the user to provide the properties to include in the response by extending the PropertyProvider class, overriding the setPropertyValue() method and passing the property provider instance to the response builder propfind() method.

WebDAVResponseBuilder Example

```
@Path("defects/{defect}")
public class DefectResource {
  @WebDAVMethod.PROPFIND
  @Consumes("application/xml")
  @Produces(application/xml")
  public Response propfindDefect(@PathParam("defect") String defected) {
    SyndFeed feed = ...
    return WebDAVResponseBuilder.propfind(feed);
  }
}
```

The propfindDefect() method is associated with the PROPFIND WebDAV Http method using the @WebDAVMethod.PROPFIND annotation.

When the propfindDefect() method is invoked, an instance of a org.apache.wink.common.model.synd.SyndFeed is created and passed to the WebDAVResponseBuilder.propfind() method in order to create the actual response.

Manual Creation

In order to create a Multistatus response manually, perform the following steps:

1. Create a new org.apache.wink.webdav.model.Multistatus instance and set its fields according to the application logic.
2. Create a new javax.ws.rs.core.Response instance, set the response code to MULTI_STATUS (207), and set its entity to be the Multistatus instance.

Return the Response instance from the resource method

This page last changed on Oct 12, 2009 by bluk.

Handler Chain - Runtime Extension

The Apache Wink runtime utilizes three Handler Chains for the complete processing of a request: Request chain, Response chain and Error chain. A handler receives a MessageContext instance for accessing and manipulating the current request information and a HandlerChain instance for advancing the chain. It is the responsibility of the handler to pass control to the next handler on the chain by invoking the doChain() method on the HandlerChain instance.

A handler may call the doChain() method several times if needed, so handlers are required to consider the possibility they will be invoked more than once for the same request.
All handler related interfaces reside in the org.apache.wink.server.handlers package.

The implementation of separate chains provides the ability to move up and down one chain before moving on to the next chain. This is particularly useful for the implementation of the JAX-RS resource-method search algorithm that includes invoking sub-resource locators, and implementing the Continued Search mode.

Handlers

There are two types of handlers:

- **System Handler -** are the handlers that implement the core engine of the Apache Wink runtime. The Apache Wink runtime will not function correctly if any of the system handlers are removed from the chain.
- **User Handler -** are the handlers that are provided by an application to customize a chains behavior and to add unique functionality to it. User handlers are not part of the core functionality of Apache Wink.

Reference
In order to customize a handler chain refer to section 2 "**Apache Wink Building Blocks**", Customization of the Handlers Chain

Message Context

The MessageContext allows the following:
Allows handlers to access and manipulate the current request informationAllows handlers to maintain a state by setting attributes on the message context, as the handlers themselves are singletons and therefore statelessAllows handlers to pass information to other handlers on the chain

Request Handler Chain

The Request Handler Chain is responsible for processing a request according to the JAX-RS specification by accepting the request, searching for the resource method to invoke, de-serializing the request entity and finally for invoking the resource method. It is responsible for invoking sub-resource locators by moving up and down the chain as needed. A Request handler is a class that implements the org.apache.wink.server.handlers.RequestHandler interface.

System Request Handlers

The following is a list of system handlers comprising the request handler chain in the order that they appear in the chain.

Handler	Description
SearchResultHandler	Responsible for throwing the search result error if there was one during the search for the resource method
OptionsMethodHandler	Generates a response for an OPTIONS request in case that there is no resource method that is associated with OPTIONS, according to the JAX-RS spec
HeadMethodHandler	Handles a response for a HEAD request in case that there is no resource method that is associated with HEAD, according to the JAX-RS spec
FindRootResourceHandler	Locates the root resource that best matches the request
FindResourceMethodHandler	Locates the actual method to invoke that matches the request, invoking sub-resource locators as needed
CreateInvocationParametersHandler	Creates the parameters of the resource method to invoke and de-serializes the request entity using the appropriate MessageBodyReader
InvokeMethodHandler	Invokes the resource method

User Request Handlers

User request handlers are inserted before the InvokeMethodHandler handler.

Reference
In order to customize a handler chain refer to section 2 "**Apache Wink Building Blocks**", Customization of the Handlers Chain

Response Handler Chain

The Response Handler Chain is responsible for handling the object returned from invoking a resource method or sub-resource method according to the JAX-RS specification. It is responsible for determining the response status code, selecting the response media type and for serializing the response entity.
A Response handler is a class that implements the **org.apache.wink.server.handlers.ResponseHandler interface.**

System Response Handlers

The following is a list of system handlers comprising the response handler chain in the order that they appear in the chain.

Handler	Description
PopulateResponseStatusHandler	Determines the response status code, according to the JAX-RS spec
PopulateResponseMediaTypeHandler	Determines the response media type, according to the JAX-RS spec

FlushResultHandler	Serializes the response entity using the appropriate MessageBodyWriter
HeadMethodHandler	Performs cleanup operations in case that there was no resource method that was associated with HEAD.

User Response Handlers

User response handlers are inserted before the FlushResultHandler handler. Apache Wink initializes the user response handler chain with the **CheckLocationHeaderHandler** handler that verifies that the "**Location**" response header is present on a response when there is a status code that requires it, for example, status code: 201.

Reference
In order to customize a handler chain refer to section 2 "**Apache Wink Building Blocks**", Customization of the Handlers Chain

Error Handler Chain

The Error Handler Chain is responsible for handling all of the exceptions that are thrown during the invocation of the Request and Response handler chains, according to the JAX-RS specification for handling exceptions. It is responsible for determining the response status code, selecting the response media type and for serializing the response entity.

An Error handler is a class that implements the org.apache.wink.server.handlers.ResponseHandler interface.

System Error Handlers

The following is a list of system handlers comprising the error handler chain in the order that they appear in the chain.

Error Handlers

Handler	Description
PopulateErrorResponseHandler	Prepares the response entity from a thrown exception according to the JAX-RS specification
PopulateResponseStatusHandler	Determines the response status code according to the JAX-RS spec
PopulateResponseMediaTypeHandler	Determines the response media type, according to the JAX-RS spec
FlushResultHandler	Serializes the response entity using the appropriate MessageBodyWriter

User Error Handlers

User error handlers are inserted before the FlushResultHandler handler.

Reference
In order to customize a handler chain refer to section 2 "**Apache Wink Building Blocks**", Customization of the Handlers Chain

Request Processing

The following details how the Apache Wink runtime performs request processing:

1. Create new instances of the three handler chains. The handlers themselves are singletons.
2. Create a new instance of a MessageContext to pass between the handlers.
3. Invoke the first handler on the Request chain.
4. Once the request chain is complete, invoke the Response chain and pass it the MessageContext that was used in the Request chain.
5. Make both chains and the MessageContext available for garbage collection.

If at any time during the execution of a Request or Response chain an exception is thrown, catch the exception, wrap it in a new MessageContext instance and invoke the Error chain to produce an appropriate response.

5.8 Link Builder

This page last changed on Oct 12, 2009 by bluk.

Link Builders

The LinkBuilders interface enables access to two types of links builders, the SystemLinksBuilder and the SingleLinkBuilder. An instance of LinkBuilders is injected into a class field or method parameter using the @Context annotation.

Upon creation, the LinkBuilders automatically detects if the target method being invoked is a resource method or a sub-resource method. The "resource" and "subResource" properties of the builder are initialized accordingly. The link builder interfaces reside in the org.apache.wink.server.utils package.

Link Builders Overview

The JAX-RS specification defines the UriBuilder interface used to construct a URI from a template, but does not specify any mechanism that can automatically generate all resource links.
Apache Wink provides the SystemLinksBuilder for automatic generation of all the alternate links to a resource, one link per every supported media type. For example, this is useful for an application that produces Atom feeds to include in the feed all the alternate representations of the resource.

Apache Wink provides a mechanism for defining if the generated links should be absolute links or relative to a base URI. For example, links embedded in an Atom feed should be as short as possible in order to optimize the payload size.

The "alt" Query Parameter

Apache Wink supports the special query parameter "alt" that is used to override the value of the request Accept header. When the link builders generate a link that specifies the "type" attribute, then the "alt" query parameter is automatically added to the generated link. This is controlled by setting the wink.addAltParam key of the configuration properties file or by calling the LinksBuilder#addAltParam() method.

> **Reference**
> For more information on the Configuration Properties File refer to section 5.1 Registration and Configuration.

System Links Builder

The SystemLinksBuilder interface enables the generation of all, or a subset of, the system links to a resource or its sub-resources. The links are generated as absolute URIs or as relative to the base URI according to the SystemLinksBuilder state, request information or the application configuration.

Example

```
@Path("defects/{id}")
public class DefectResource {
@GET
@Produces("application/atom+xml")
public SyndEntry getAtom() {    ...  }
@GET
@Produces("application/json")
public JSONObject getJson() {    ...  }
}
@GET
@Produces("application/xml")
```

```
public Defect getXml(@Context LinkBuilders linkBuilders) {     SystemLinksBuilder builder =
 linkBuilders.systemLinksBuilder();     ListsystemLinks = builder.build(null);     ...   }
}
```

Explanation

The DefectResource#getXml() method is invoked when a GET request for application/xml is made to /defects/3. The Apache Wink runtime injects an instance of LinkBuilders to the linkBuilder parameter and a new instance of a SystemLinksBuilder is created by invoking the systemLinksBuilder() method.
The call to the build() method of the SystemLinksBuilder generates three alternate links to the DefectResource and the self link:

- <link rel="self" href="/defects/3"/>
- <link rel="alternate" type="application/json" href="/defects/3"/>
- <link rel="alternate" type="application/xml" href="/defects/3"/>
- <link rel="alternate" type="application/xtom+xml" href="/defects/3"/>

Single Link Builder

The SingleLinkBuilder interface enables the generation of a single link referencing a resource or a sub-resource, allowing the specification of the 'rel' and 'type' attributes of the generated link. The links are generated as absolute URIs or as relative to the base URI according to the SingleLinkBuilder state, request information or the application configuration.
Generating Absolute or Relative Links
The link builders generate absolute or relative links based on the following algorithm:

1. Use the value that was passed to the relativize() method of the builder.
2. If the relativize() method was not called, then use the value of the "relative-urls" query parameter from the request. The value must be either true or false.
3. If the request does not contain the "**relative-urls**" query parameter, then use the value of the wink.defaultUrisRelative key set in the application configuration properties file. The value must be either true or false.
4. If the configuration key does not exist, then use true.

Reference
For more information on the Configuration Properties File refer to section 5.1 Registration and Configuration.

5.9 Assets

This page last changed on Oct 13, 2009 by michael.

Assets

An Asset is a special entity that is returned by a resource method or is injected into a resource method as an entity parameter. The asset is used for retrieving the actual request entity or response entity.
The purpose of an asset is to act as a container of an entity data model while providing the transformation methods of the data model into data models of other representations.
Asset classes are POJOs, annotated with the @Asset annotation, that have any number of entity methods.

When an asset instance is returned from a resource method or is set as the entity on a Response instance, it is used by the Apache Wink runtime to retrieve the actual response entity by invoking the appropriate entity-producing method of the asset.

Reference
For more information on Entity-Producing Methods refer to section Entity Producing Methods.

When an asset is the entity parameter of a resource method, it is used by the Apache Wink runtime to set the actual request entity by invoking the appropriate entity-consuming method of the asset.

Assets Overview

A typical application exposes each resource in a number of representations. Some form of data model usually backs the resource, and the application business logic relies on the manipulation of that data model. The application will most likely expose resource methods allowing the consumption of the data model in more than one representation (for example Atom and XML) and the production of the data model in other representation (for example Atom, XML and JSON).

According to the JAX-RS specification, the optimal method for implementing a resource is one that consumes and produces an application data model and makes use of a different provider for every media type.

For example, if a resource implements methods that consume and produce a "Defect" bean, then a provider must be implemented for each representation of the "Defect" (Atom, XML and JSON). However, there are times that the transformation of the application data model into a representation requires information that may only be available to the resource but is unavailable to a provider (for example, a connection to the Database).

There are several solutions for dealing with the problem of a provider not having sufficient information to perform application data transformations. The following is a description of two possible solutions:

- Passing the information as members on the resource and accessing the resource from the provider via the UriInfo context.

This solution is only plausible if the resource scope is "per request" and does not work if the resource is a singleton.

- Passing the information from the resource to the provider via the attributes of the HttpServletRequest.

This solution is only plausible when the application is deployed in a JEE container and is not the optimal solution.
In addition to the previously mentioned problem, the creation of a provider for every data model per media type may result in the inflation of providers in the system, causing the provider selection algorithm to consider a large set of potential providers.

As a result, the selection of the actual provider from the set of potential providers is non-deterministic, because the selection between them is undefined.

Performance Degradation
An additional side effect of provider inflation is performance degradation.

The use of an asset solves the problem of passing information between a resource and a provider and reduces the amount of registered providers in the system.

Lifecycle

Resource methods can use an asset as a response entity and as a request entity. The Apache Wink runtime applies different lifecycles for each case.

Response Entity Asset

The lifecycle of an asset as a response entity is as follows:

1. The application creates and returns the asset from the resource method.
2. The appropriate entity-producing method is invoked by the Apache Wink runtime to retrieve the actual response entity.
3. The appropriate message body writer as obtained from the Providers#getMessageBodyWriter() method serializes the entity obtained at the previous step.
4. The asset is made available for garbage collection.

Request Entity Asset

The lifecycle of an asset as a request entity is as follows:

1. An asset class is instantiated by the Apache Wink runtime by invoking the asset default constructor. Note that this implies that the asset class must have a public default constructor.
2. The appropriate message body reader as obtained from the Providers#getMessageBodyReader() method is invoked by the Apache Wink runtime to read the request entity.
3. The appropriate entity-consuming method is invoked on the asset to populate the asset with the request entity.
4. The asset is injected into the resource method as the entity parameter.
5. The asset is made available for garbage collection after returning from the resource method.

Asset Entity Methods

Asset Entity methods are the public methods of an asset annotated with either @Consumes or @Produces annotation. Annotating a method with both @Consumes and @Produces annotations is not supported and may result in unexpected behavior.

Entity Producing Methods

An Entity Producing Method is a public asset method annotated with the @Produces annotation, designating it to produce the actual response entity. Such methods produce an entity only for the media types declared in the @Produces annotation. Note that under this definition, wildcard ("**/**") is allowed.
The Apache Wink runtime will not invoke an entity-producing method whose effective value of @Produces does not match the request Accept header

Entity Consuming Methods

An Entity Consuming Method is a public asset method annotated with the @Consumes annotation, designating it to consume the actual request entity for populating the asset. Such methods consume an entity only for the media types declared in the @Consumes annotation. Note that under this definition, wildcard ("**/**") is allowed.
The Apache Wink runtime will not invoke an entity-consuming method whose effective value of @Consumes does not match the request Content-Type header.

Parameters

Asset Entity methods support the same parameter types as JAX-RS specifies for a resource method.

Return Type

Entity methods may return any type that is permissible to return from a resource method.

Exceptions

Exceptions thrown from an entity method are treated as exceptions thrown from a resource method.

Annotation Inheritance

The @Produces and @Consumes annotations are not inherited when an asset sub-class overrides an asset entity method. Asset sub-classes must re-declare the @Produces and @Consumes annotations for the overriding method to be an entity method.

Entity Method Matching

Asset classes are handled by the AssetProvider which is a JAX-RS provider that is capable of consuming and producing all media types.

Reference
For more information on Asset Providers refer to section 7.7 Assets Provider.

Request Entity Matching

The following points describe the process of selecting the asset entity-consuming method to handle the request entity. This process occurs during the invocation of the **AssetProvider#isReadable()** method.

1. Collect all the entity-consuming methods of the asset. These are the public methods annotated with @Consumes annotation.
2. Sort the collected entity-consuming methods in descending order, where methods with more specific media types precede methods with less specific media types, following the rule n/m > n/* > **/**.
3. Select the first method that supports the media type of the request entity body as provided to the AssetProvider#isReadable() method, and return true.
4. If no entity-consuming method supports the media type of the request entity body, return false. The Apache Wink runtime continues searching for a different provider to handle the asset as a regular entity.

Response Entity Matching

The following points describe the process of selecting an entity-producing method to produce the actual response entity. The following process occurs during the invocation of the **AssetProvider#isWriteable()** method.

1. Collect all the entity-producing methods of the asset. These are the public methods annotated with @Produces annotation.
2. Sort the collected entity-producing methods in descending order, where methods with more specific media types precede methods with less specific media types, following the rule n/m > n/* > **/**.
3. Select the first method that supports the media type of the response entity body as provided to the AssetProvider#isWriteable()method and return true.
4. If no entity-producing method supports the media type of the response entity body, return false. The Apache Wink runtime continues searching for a different provider to handle the asset as a regular entity.

Asset Example

The following example illustrates the use of an asset. The "**Defect**" bean is a JAXB annotated class.

DefectAsset Class

The DefectAsset class is the asset backed by an instance of a "Defect" bean.

```
@Asset
public class DefectAsset {
    public Defect defect;
    public DefectAsset(Defect defect) {
        this.defect = defect;
    }
    @Produces("application/xml")
    public Defect getDefect() {
        return this.defect;
    }
    @Produces("text/html")
    public String getDefectAsHtml() {
        String html = ...;
        return html;
    }

    @Produces("application/atom+xml")
    public AtomEntry getDefectAsAtom() {
        AtomEntry entry = ...;
        return entry;
    }
    @Consumes("application/xml")
    public void setDefect(Defect defect) {
        this.defect = defect;
    }
}
```

DefectResource Class

The DefectResource class is a resource that is anchored to the URI path "**defects/{id}**" within the Apache Wink runtime.

```
@Path("defects/{id}")
public class DefectResource {
    @GET
    public DefectAsset getDefect(@PathParam("id") String id) {
        return new DefectAsset(defects.get(id));
    }
    @PUT
    public DefectAsset updateDefect(DefectAsset defectAsset,
                                    @PathParam("id") String id) {
        defects.put(id, defectAsset.getDefect());
        return defectAsset;
    }
}
```

Scenario Explanation 1

1. A client issues an HTTP GET request with a URI="/defects/1" and Accept Header= "application/xm

2. The Apache Wink runtime analyzes the request and invokes the DefectResource#getDefect() resource method.
3. The DefectResource#getDefect() resource method creates an instance of DefectAsset and populates it with defect "1" data
4. The DefectResource#getDefect() resource method returns the DefectAsset instance back to Apache Wink runtim
5. The Apache Wink runtime analyzes the asset and invokes the DefectAsset#getDefect() entity-producing method to obtain the reference to the "Defect" bean.
6. The "Defect" bean is serialized by Apache Wink runtime as an XML using the appropriate provider.

Scenario Explanation 2

1. A Client issues an HTTP GET request with a URI="/defects/1" and Accept Header= "text/html"
2. The Apache Wink runtime analyzes the request and invokes the DefectResource#getDefect() resource metho
3. The DefectResource#getDefect() resource method creates an instance of DefectAsset and populates it with defect "1" data.
4. The DefectResource#getDefect() method returns the populated asset back to the Apache Wink runtime.
5. The Apache Wink runtime analyzes the asset and invokes the DefectAsset#getDefectAsHtml() entity-producing method in order to obtain the reference to the "Defect" bean.
6. The "Defect" is serialized by Apache Wink runtime as an HTML using the appropriate provider.

Scenario Explanation 3

1. A Client issues an HTTP PUT request with a URI="/defects/1" and Accept Header= "text/html"
2. The Apache Wink runtime analyzes the request and invokes the DefectResource#updateDefect() method with an instance of DefectAsset populated with the request entity.* A DefectAsset is instantiated by the Apache Wink runtime
3. The DefectAsset#setDefect() entity-consuming method is invoked in order to populate the DefectAsset with the defect data.

5.10 Admin Views

This page last changed on Oct 13, 2009 by michael.

Administration Views

Apache Wink provides administration views in order to help developers better understand the services that the REST application expose. There are two administration views, the "**application resource XML view**" and the "**resource registry XML view**".

Application Resources XML View

The application resource XML view shows the way the application is exposed to the user, it exposes the REST resources with their URI templates, the HTTP methods that are supported by the resources and the consume/produces MimeType supported by each method. This view can be used as a base for the service documentation.

Resource Registry XML View

The resource registry XML view shows the way that the application is developed, it is similar to the "**Application resources XML view**" but it also exposes the classes that implement resources and their priority in the registry. This view can be useful for debugging.

Configuration

By default these views are disabled, in order to activate them register the "**AdminServlet**" implemented by the **org.apache.wink.server.internal.servlet.AdminServletin** the web.xml file. No init parameters are required.

Example

The following code snippet is an example of a web application descriptor file.

```
<servlet>
      <servlet-name>restSdkAdmin</servlet-name>
      <servlet-class>org.apache.wink.server.internal.servlet.AdminServlet</servlet-class>
</servlet>

<servlet-mapping>
      <servlet-name>restSdkAdmin</servlet-name>
      <url-pattern>/admin</url-pattern>
</servlet-mapping>
```

6 Apache Wink Client

This page last changed on Oct 14, 2009 by michael.

Apache Wink Client

The following section describes the Apache Wink Client and provides a detailed description of the Apache Wink Client component and its functionality.

Contents

Apache Wink Client Overview

The Apache Wink Client is an easy-to-use client, providing a high level Java API for writing clients that consume HTTP-based RESTful Web Services. The Apache Wink Client utilizes JAX-RS concepts, encapsulates Rest standards and protocols and maps Rest principles concepts to Java classes, which facilitates the development of clients for any HTTP-based Rest Web Services.

The Apache Wink Client also provides a Handlers mechanism that enables the manipulation of HTTP request/response messages.

Main Features

The Apache Wink Clients main features are as follows:

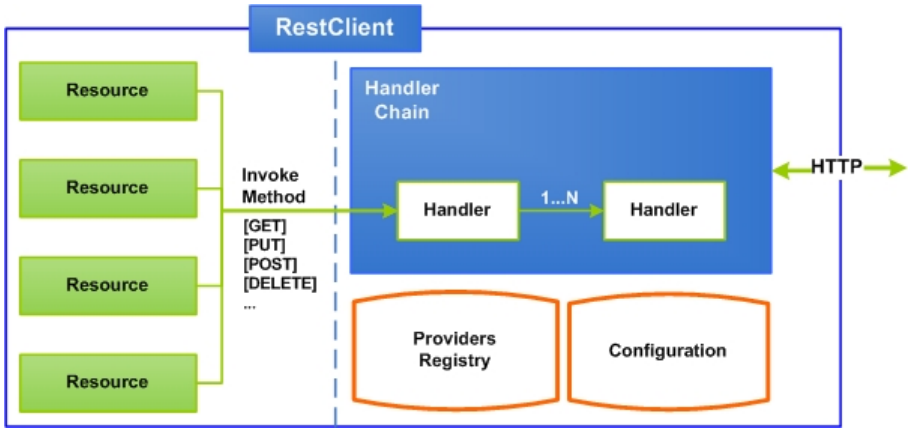
- Utilizes JAX-RS Providers for resource serialization and deserialization
- Provides Java object models for Atom, Json, RSS, APP, CSV, Multipart and OpenSearch along with providers to serialize and deserialize these models
- Uses the JDK HttpUrlConnection as the underlying Http transport
- Allows for the easy replacement of the underlying Http transport
- Provides a Handlers mechanism for manipulation of HTTP request and response messages

Supports

- Http proxy
- SSL

Apache Wink High Level Client Architecture Overview

The following diagram illustrates the high-level architecture of the Apache Wink Client.



The RestClient class is the Apache Wink Client entry point and is responsible for holding different configuration options and the provider registry.

The RestClient is used to create instances of the Resource class. The Resource class represents a web resource associated with a specific URI and is used to perform uniform interface operations on the resource it represents. Every method invocation goes through a user defined handlers chain that enables for manipulation of the request and response.

6.1 Getting Started with Apache Wink Client

This page last changed on Oct 13, 2009 by michael.

Getting Started with the Apache Wink Client

The following section details the getting started examples that demonstrate how to write a simple client that consume RESTful Web Services with the Apache Wink Client.

GET Request

The following example demonstrates how to issue an Http GET request.

```
// create the rest client instance
1  RestClient client = new RestClient();

// create the resource instance to interact with
2  Resource resource = client.resource("http://services.com/HelloWorld");
// perform a GET on the resource. The resource will be returned as plain text
3  String response = resource.accept("text/plain").get(String.class);
```

Explanation

The RestClient is the entry point for building a RESTful Web Service client. In order to start working with the Wink Client, a new instance of RestClient needs to be created, as the example shows in line 1 of the example. A new Resource is then created with the given URI, by calling the RestClient#resource() method as appears in line 2.

Finally, the Resource#get() method is invoked in order to issue an Http GET request as appears in line 3. Once the Http response is returned, the client invokes the relevant provider to desterilizes the response in line 3.

POST Request

The following example demonstrates how to issue an Http POST request.

```
// create the rest client instance
1  RestClient client = new RestClient();

// create the resource instance to interact with
2  Resource resource = client.resource("http://services.co");

// issue the request
3  String response = resource.contentType("text/plain").accept("text/plain").post(String.class, "foo");
```

Explanation

The POST Request example demonstrates how to issue a simple Http POST request that sends and receives resources as strings.

First, a new instance of a Resource is created through the RestClient. The Http POST request is then issued by specifying the request and response media types and the response entity type (String.class).

POST Atom Request

The following example demonstrates how to issue an Http POST request that sends and receives atom entries.

```
// create the rest client instance
1  RestClient client = new RestClient();

// create the resource instance to interact with
2  Resource resource = client.resource("http://services.co");

3  AtomEntry request = getAtomEntry();

// issue the request
4  AtomEntry response = resource.contentType("application/atom+xml").accept("application/atom
+xml").post(AtomEntry.class, request);
```

Explanation

The Apache Wink Client provides an object model for Atom (atom feed and atom entry), and supplies out-of-the-box providers that enable sending and receiving atom feeds and entries.

Using ClientResponse

The following example demonstrates how to use the ClientResponse object in order to de-serialize the response entity.

```
// create the rest client instance
1  RestClient client = new RestClient();

// create the resource instance to interact with
2  Resource resource = client.resource("http://services.co");

// issue the request
3  ClientResponse response = resource.accept("text/plain").get();

// deserialize response
4  String responseAsString = response.getEntity(String.class);
```

Explanation

If the response entity type is not provided when invoking the Resource#get() method that appears in line 3, the response will be returned as the raw ClientResponse. In order to trigger the response deserialization mechanism, the ClientResponse#getEntity() method needs to be invoked as it appears in line 4 with the required response entity type.

6.2 Configuring the Apache Wink Client

This page last changed on Apr 17, 2010 by bluk.

Client Configuration

The RestClient configuration is performed by using the ClientConfig class. An instance of the configuration class is passed to the constructor of the RestClient when constructing a new RestClient.

The following options can be configured in the RestClient:

- Custom providers via JAX-RS Application
- Handler chain
- Proxy host and port
- Connect and read timeouts
- Redirect

Handler Configuration

The following example demonstrates how to register a custom handler.

```
1  ClientConfig config = new ClientConfig();
// Create new JAX-RS Application
2  config.handlers(new DummyHandler());
// create the rest client instance
3  RestClient client = new RestClient(config);
// create the resource instance to interact with
4  Resource resource = client.resource("http://services.com/HelloWorld");
// perform a GET on the resource
// the resource will be returned as plain text
5  String response = resource.accept("text/plain").get(String.class);
```

Explanation

First, a new instance of a ClientConfig is created as it appears in line 1. Then the new handler is added to the handlers chain by invoking the handlers() method on the ClientConfig instance as it appears in line 2. Finally, a new instance of a RestClient is created with this configuration as it appears in line 3.

org.apache.wink.client.handlers.BasicAuthSecurityHandler

You can configure a simple BasicAuthSecurityHandler to use basic authentication when accessing resources from the client.

```
ClientConfig config = new ClientConfig();<br/>
BasicAuthSecurityHandler basicAuthHandler = new BasicAuthSecurityHandler();
basicAuthHandler.setUserName("foo");
basicAuthHandler.setPassword("bar");
config.handlers(basicAuthHandler);<br/>
// create the rest client instance<br/>
RestClient client = new RestClient(config);<br/>
// create the resource instance to interact with Resource<br/>
resource = client.resource("http://localhost:8080/path/to/resource");<br/>
```

Explanation

The BasicAuthSecurityHandler will attempt to access the resource without security information first. If the request is successful, then the BasicAuthSecurityHandler is like a no-op. However, if it receives a 401, then the username and password will be added to the request, and the request will be sent again.

Gzip Input Stream Adapter

The following code snippet is an example of an implementation of a Gzip input stream adapter.

```
class GzipInputAdapter implements InputStreamAdapter{
       public InputStream adapt(InputStream is,
                           ClientResponse response) {
     String header = response.getHeaders().getFirst("Content-Encoding");
     if (header != null && header.equalsIgnoreCase("gzip")) {
       return new GZIPInputStream(is);
     }
     return is;
   }}
```

Explanation

The Gzip input stream adapter is responsible for wrapping the input stream with the Gzip input stream.

Gzip Output Stream Adapter

The following code snippet is an example of an implementation of a Gzip output stream adapter.

```
class GzipOutputAdapter implements OutputStreamAdapter {
    public OutputStream adapt(OutputStream os,
                         ClientRequest request) {
       request.getHeaders().add("Content-Encoding", "gzip");
       return new GZIPOutputStream(os);
    }}
```

Explanation

The Gzip output stream adapter is responsible for wrapping the output stream with the Gzip output stream.

Custom Provider Configuration

The following example demonstrates how to register a custom entity provider.

```
1  ClientConfig config = new ClientConfig();
   // Create new JAX-RS Application
2    Application app = new Application() {
     @Override
      public Set<Class<?>> getClasses() {
        HashSet<Class<?>> set = new HashSet<Class<?>>();
        set.add(FooProvider.class);
        return set;}};
3  conf.applications(app);
// create the rest client instance
4  RestClient client = new RestClient(config);
```

```
// create the resource instance to interact with
5  Resource resource = client.resource("http://services.com/HelloWorld");
// perform a GET on the resource. the resource will be returned as plain text
6  String response = resource.accept("text/plain").get(String.class);
```

Explanation

First, a new instance of ClientConfig is created as it appears in line 1. Then a new anonymous Application is instantiated and set on the ClientConfig as it appears in line 2 and 3. Finally, a new instance of a RestClient is created with this configuration as it appears in line 4.

6.3 Input and Output Stream Adapters

This page last changed on Oct 13, 2009 by michael.

Input and Output Stream Adapters

The Apache Wink Client provides the ability to manipulate raw Http input and output entity streams through the InputStreamAdapter and the OutputStreamAdapter interfaces. This is useful for modifying the input and output streams, regardless of the actual entity, for example when adding compression capabilities.
The adapt() method of the output stream adapter is called before the request headers are committed, in order to allow the adapter to manipulate them.

The adapt() method of the input stream adapter is called after the response status code and the headers are received in order to allow the adapter to behave accordingly.

Stream Adapters Example

The following example demonstrates how to implement input and output adapters.

Gzip Handler

The following code snippet is an example of adding a input and output adapters in the Gzip handler.

```
public class GzipHandler implements ClientHandler {
   public ClientResponse handle(ClientRequest request,
                           HandlerContext context) {
       request.getHeaders().add("Accept-Encoding", "gzip");
       context.addInputStreamAdapter(new GzipInputAdapter());
       context.addOutputStreamAdapter(new GzipOutputAdapter());
       return context.doChain(request);
  }}
```

Explanation

The Gzip handler creates instances of the **GzipInputAdapter** and the **GzipOutputAdapter** and adds them to the stream adapters of the current request by invoking the **addInputStreamAdapter()** method and the **addOutputStreamAdapter()** method on the HandlerContext instance.

Gzip Input Stream Adapter

The following code snippet is an example of an implementation of a Gzip input stream adapter.

```
class GzipInputAdapter implements InputStreamAdapter{
       public InputStream adapt(InputStream is,
                             ClientResponse response) {
     String header = response.getHeaders().getFirst("Content-Encoding");
     if (header != null && header.equalsIgnoreCase("gzip")) {
       return new GZIPInputStream(is);
     }
     return is;
   }}
```

Explanation

The Gzip input stream adapter is responsible for wrapping the input stream with the Gzip input stream.

Gzip Output Stream Adapter

The following code snippet is an example of an implementation of a Gzip output stream adapter.

```
class GzipOutputAdapter implements OutputStreamAdapter {
    public OutputStream adapt(OutputStream os,
                              ClientRequest request) {
        request.getHeaders().add("Content-Encoding", "gzip");
        return new GZIPOutputStream(os);
    }}
```

Explanation

The Gzip output stream adapter is responsible for wrapping the output stream with the Gzip output stream.

7 Apache Wink Representations

This page last changed on Apr 20, 2010 by shivakumar.

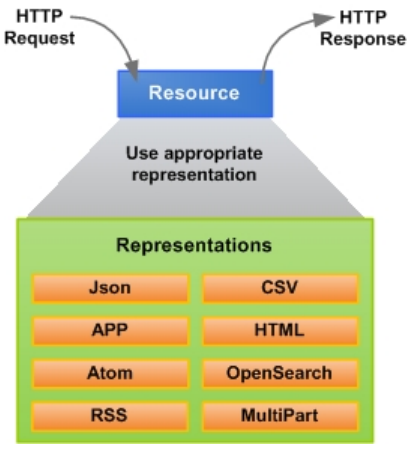
Apache Wink Representations

In addition to the JAX-RS standard representations Apache Wink provides an expanded set of representations that can be used by an application to expose resources in a rich set of representations. The following section provides information about the representations that are supported by Apache Wink.

Contents

Apache Wink Representations Request Flow

The following diagram illustrates the client request flow for a resource.



A resource is an abstract entity and as such can not be transmitted between peers. When a client is required to send or receive a resource it must use a representation of that resource. The resource representation is a specific formatting of the resource data. The diagram illustrates that a request for a resource is in fact a request for a specific representation of that resource.

Apache Wink implements resource representations through the use of providers for java objects as defined by the JAX-RS specification.

Apache Wink Providers

In addition to JAX-RS standard providers (**refer to section 4.2 of the JAX-RS specification**), Apache Wink provides a set of complementary providers. The purpose of these providers is to provide mapping services between various representations for example Atom, APP, RSS, OpenSearch, CSV, JSON and HTML, and their associated Java data models.

The Apache Wink providers are pre-registered and delivered with the Apache Wink runtime along with the JAX-RS standard providers.

Apache Wink provides an additional method for defining the life cycle of a provider via the use of the @Scope annotation and a way to define the providers priorities.

Scoping

The JAX-RS specification defines by default, that a singleton instance of each provider class is instantiated for each JAX-RS application. Apache Wink fully supports this requirement and in addition provides a "Prototype" lifecycle, which is an instance per-request lifecycle.
Prototype means that a new instance of a provider class is instantiated for each request. The @Scope annotation (section#0) is used on a provider class to specify its lifecycle. The lifecycle of a provider that does not specify the @Scope annotation defaults to the singleton lifecycle.

Prototype Example

The following example shows how to define a provider with a prototype lifecycle.

```
@Scope(ScopeType.PROTOTYPE)
@Provider
public class MyProvider implements MessageBodyReader<String>{
   ...
}
```

Singleton Example 1

The following example shows how to define a provider with a singleton lifecycle.

```
@Scope(ScopeType.SINGLETON)
@Provider
public class MyProvider implements MessageBodyReader<String>{
   ...
}
```

Singleton Example 2

The following example shows that when the @Scope annotation is not used, the provider will be a singleton, as per the JAX-RS specification.

```
@Provider
public class MyProvider implements MessageBodyReader<String>{
   ...
}
```

Priority

Apache Wink provides a method for setting a priority for a provider.

Reference
For more information on Provider Priorities refer to section 5.1 Registration and Configuration.

7.1 JSON

This page last changed on Apr 20, 2010 by shivakumar.

JSON Providers

Apache Wink provides a set of providers that are capable of serializing a number of data models into JSON representations. There are currently 3 Apache Wink extensions that provide JSON support. Each has unique features that may make one more suitable for your particular application.

wink-json-provider (org.json)

The wink-json-provider extension is provided in the binary distribution and uses the JSON.org classes to provide JSON support. Include the wink-json-provider-<VERSION>.jar in the classpath and the providers will automatically be registered. You will also need the org.json JAR which is provided in the ext/wink-json-provider/lib folder.

org.apache.wink.providers.json.JsonProvider

Handles reading and writing of org.json.JSONObject classes for the application/json and application/javascript media types.

	Supported	Media Types	Entity
Read	Yes	application/json , application/javascript	org.json.JSONObject
Write	Yes	application/json , application/javascript	org.json.JSONObject

org.apache.wink.providers.json.JsonArrayProvider

Handles reading and writing of org.json.JSONArray classes for the application/json and application/javascript media types.

	Supported	Media Types	Entity
Read	Yes	application/json , application/javascript	org.json.JSONArray
Write	Yes	application/json , application/javascript	org.json.JSONArray

org.apache.wink.providers.json.JsonJAXBProvider

Handles reading and writing of JAXBElement and JAXB annotated classes for the application/json and application/javascript media types.

	Supported	Media Types	Entity
Read	Yes	application/json , application/javascript	JAXB object, JAXBElement<?>>>
Write	Yes	application/json , application/javascript	JAXB object, JAXBElement<?>>>

Producing and Consuming JSON Example

The following example demonstrates the usage of a JSON provider for reading and writing JSON representations.

```
@GET
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public JSONObject postJSON(JSONObject requestJSON) {
    String property = requestJSON.getString("property");
    JSONObject jobj = new JSONObject();
    return jobj;
}

/* Book is a JAXB annotated class */

@GET
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Book postJSONBook(Book requestBookEntity) {
    String title = requestBookEntity.getTitle();
    /* other code */
    Book response = new Book();
    return response;
}
```

wink-jettison-provider (org.codehaus.jettison)

The wink-jettison-provider extension is provided in the binary distribution and uses the Jettison code to provide JSON support. Include the wink-jettison-provider-<VERSION>.jar in the classpath and the providers will automatically be registered. You will also need the Jettison library JARs which are provided in the ext/wink-jettison-provider/lib folder.

By default, reading is currently disabled due to potential issues with the reader. You can enable it by calling setUseAsReader(boolean) on each provider and registering as a singleton in the JAX-RS Application sub-class.

org.apache.wink.providers.jettison.JettisonJAXBElementProvider

Handles reading and writing of JAXBElement classes for the application/json media type.

	Supported	Media Types	Entity
Read	Yes	application/json	JAXBElement<?>>>
Write	Yes	application/json	JAXBElement<?>>>

org.apache.wink.providers.jettison.JettisonJAXBElementProvider

Handles reading and writing of JAXB annotated classes for the application/json media type.

	Supported	Media Types	Entity
Read	Yes	application/json	JAXB object
Write	Yes	application/json	JAXB object

Example

The following examples demonstrates the usage of a Jettison provider for producing and consuming JSON.

Jettison Provider Registration

The following code example demonstrates the way to register a Jettison provider within a JAX-RS application.

```
public class MyApp extends Application {
   public Set getClasses() {
      Set s = new HashSet();
      s.add(MyResource.class);
      return s;
   }

   public Set<Object> getSingletons() {
      Set s = new HashSet();
      JettisonJAXBProvider jaxbProvider = new JettisonJAXBProvider();
      jaxbProvider.setUseAsReader(true);
      return s;
   }
}
```

Producing and Consuming JSON

The following code example demonstrates the reading and writting of JAXB objects into a JSON representation by using a Jettison provider.

```
/* Book is a JAXB annotated class */

@GET
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Book postJSONBook(Book requestBookEntity) {
   String title = requestBookEntity.getTitle();
   /* other code */
   Book response = new Book();
   return response;
}
```

Jackson JSON Processor

Jackson JSON Processor may also suit your needs and can be used. They provide their own JAX-RS entity provider. See their documentation for more information.

7.2 APP

This page last changed on Oct 13, 2009 by michael.

Atom Publishing Protocol (AtomPub) Overview

The Atom Publishing Protocol (AtomPub) is an application-level protocol for publishing and editing Web resources. The protocol is based on HTTP transport of Atom-formatted representations. The Atom format is documented in the Atom Syndication Format.

Data Model

Apache Wink provides an Atom Publishing Protocol data model for producing Service Documents (application/atomsvc+xml) and Categories Documents (application/atomcat+xml). All of the model classes are located under the org.apache.wink.common.model.app package.

Important Note
The APP data model can also be used to produce Service and Categories documents in HTML (text/html) and JSON (application/json) formats. For more details regarding HTML see section HTML (TBD). For JSON format see section (TBD)

APP Service Document Support

The following table shows the APP service document data models and the representations in which it can be serialized and de-serialized.

	Supported	Media Types	Data Model	Provider Registration
Read	Yes	application/ atomsvc+xml	org.apache.wink.common .model.app.AppService	Not required. Registered by default
Write	Yes	application/ atomsvc+xml	org.apache.wink.common .model.app.AppService	Not required. Registered by default

APP Categories Document Support

The following table shows the APP Catagories document data models and the representations in which it can be serialized and de-serialized.

	Supported	Media Types	Data Model	Provider Registration
Read	Yes	application/ atomcat+xml	org.apache.wink.common .model.app.AppCategories	Not required. Registered by default
Write	Yes	application/ atomcat+xml	org.apache.wink.common .model.app.AppCategories	Not required. Registered by default

Producing an APP Service Document Example

The following code example demonstrates the creation of an APP Service Document.

```
@GET
 @Produces(MediaTypeUtils.ATOM_SERVICE_DOCUMENT)
 public AppService getServiceDocument() {
    AppService serviceDocument = new AppService();
    List<AppWorkspace> workspaces = serviceDocument.getWorkspace();
    AppWorkspace workspace1 =  new AppWorkspace();
    workspace1.setTitle(new AtomText("Workspace1"));
    List<AppCollection> collections = workspace1.getCollection();
    AppCollection collection = new AppCollection();
    collection.setTitle(new AtomText("Collection1"));

    collections.add(collection);
    workspaces.add(workspace1);
    ...
    return serviceDocument;
 }
```

Explanation

AppService class is JAXB annotated POJO. An instance of an AppService class is created, populated and returned by the resource. A generic JAXB provider is used to serializes this class into an XML representation.

7.3 Atom

This page last changed on Oct 15, 2009 by shivakumar.

Atom Syndication Format Overview

Atom is an XML-based document format that describes lists of related information known as "feeds". Feeds are composed of a number of items, known as "entries", each with an extensible set of attached metadata. For example, each entry has a title. The primary use case that Atom addresses is the syndication of Web content such as weblogs and news headlines to Web sites as well as directly to user agents.

Data Model

Apache Wink provides an Atom Syndication Format data model for consuming and producing Atom Feeds and Atom Entries (application/atom+xml). All of the model classes are located under the org.apache.wink.common.model.atom and org.apache.wink.common.model.synd packages.

Important Note
The Atom Syndication Format data model can also be used to produce Atom Feeds and Atom Entries in HTML (text/html) and JSON (application/json) formats. For more details regarding HTML see section HTML (TBD). For JSON format see section (TBD)

Atom Feed Support

The following table shows the Atom Feed data models and the representations in which it can be serialized and de-serialized.

	Supported	Media Types	Data Model	Provider registration
Read	Yes	application/atom+xml	org.apache.wink.common.model.atom.AtomFeed org.apache.wink.common.model.synd.SyndFeed	Not required. Registered by default
Write	Yes	application/atom+xml	org.apache.wink.common.model.atom.AtomFeed org.apache.wink.common.model.synd.SyndFeed	Not required. Registered by default

Atom Entry Support

The following table shows the Atom Entry data models and the representations in which it can be serialized and de-serialized.

	Supported	Media Types	Data Model	Provider registration
Read	Yes	application/atom+xml	org.apache.wink.common.model.atom.AtomEntry org.apache.wink.common.model.synd.SyndEntry	Not required. Registered by default

Write	Yes	application/atom +xml	org.apache.wink .common.model .atom.AtomEntry org.apache.wink .common.model .synd.SyndEntry	Not required. Registered by default

Examples

The following code example demonstrates reading and writing of Atom Feeds and Atom Entries.

Producing Atom Feed

The following code example demonstrates the creation of an Atom Feed.

```
    @GET
    @Produces(MediaType.APPLICATION_ATOM_XML)
    public AtomFeed getFeed() {
        AtomFeed feed = new AtomFeed();
        feed.setId("http://example.com/atomfeed");
        feed.setTitle(new AtomText("Example"));
        feed.setUpdated(new Date());
        AtomLink link1 = new AtomLink();
        ...

        return feed;
    }
}
```

Consuming Atom Feed

The following code example demonstrates the consumption of an Atom Feed.

```
    @POST
    @Consumes(MediaType.APPLICATION_ATOM_XML)
    public void setFeed(AtomFeed feed) {
        ...

        return;
    }
```

Producing Atom Entry

The following code example demonstrates the creation of an Atom Entry.

```
    @GET
    @Produces(MediaType.APPLICATION_ATOM_XML)
    public AtomEntry getEntry() {
        AtomEntry entry = new AtomEntry();
        entry.setId("http://example.com/entry");
        entry.setTitle(new AtomText("Web Demo"));
        entry.getLinks().add(link2);
        entry.setUpdated(new Date());
        entry.setPublished(new Date());
        ...
```

```
        return entry;
    }
```

Consuming Atom Entry

The following code example demonstrates the consumption of an Atom Entry.

```
    @POST
    @Consumes(MediaType.APPLICATION_ATOM_XML)
    public void setEntry(AtomEntry entry) {
        ...

        return;
    }
```

7.4 RSS

This page last changed on Apr 21, 2010 by shivakumar.

RSS Data Model

RSS (Really Simple Syndication) is an XML-based document format for the syndication of web content such as weblogs and news headlines to Web sites as well as directly to user agents. Apache Wink supports the RSS 2.0 specification.

RSS Data Model Overview

Apache Wink provides an RSS data model for consuming and producing RSS Feeds (application/xml). All of the model classes are located under **org.apache.wink.common.model.rss** package.

RSS Feed Support

The following table shows the RSS Feed data models and the representations in which it can be serialized and de-serialized.

	Supported	Media Types	Data Model	Provider registration
Read	Yes	application/xml	org.apache.wink .common.model .rss.RssFeed	Not required. Registered by default
Write	Yes	application/xml	org.apache .wink.common .model.rss .RssFeed	Not required. Registered by default

Examples

The following code example demonstrates reading and writing of RSS Feeds.

Producing RSS Feed

The following code example demonstrates the creation of an RSS Feed.

```
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public RssFeed getFeed() {
        RssFeed rss = new RssFeed();

        RssChannel channel = new RssChannel();
        channel.setTitle("Liftoff News");
        channel.setLink("http://liftoff.msfc.nasa.gov");
        channel.setDescription("Liftoff to Space Exploration.");
        channel.setPubDate(new Date().toString());

        RssItem item = new RssItem();
        item.setTitle("Star City");
        item.setLink("http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp");
        item.setDescription("How do Americans get ready to work with Russians aboard the International Space
Station?");
        channel.getItems().add(item);

        ...
```

```
        rss.setChannel(channel);
        return rss;
    }
```

Consuming RSS Feed

The following code example demonstrates the consumption of an RSS Feed.

```
    @POST
    @Consumes(MediaType.APPLICATION_XML)
    public void setFeed(RssFeed feed) {
        ...

        return;
    }
```

RSS Mapped into Syndication Object Model

Starting with v1.1 of Apache Wink, the RSS object model has been mapped into Apache Wink's Syndication Object Model. This enables common Synd APIs (available in **org.apache.wink.common.model.synd** package) to be used for consuming/producing both RSS and Atom documents.

Mapping between RSS and Atom:

The following table illustrates the mapping between RSS and Syndication Object Model (which in turn is same as Atom model):

RSS document	Mapped into Atom
<?xml version="1.0"?> <rss version="2.0"> <channel>	<?xml version="1.0" encoding="utf-8"?> <feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en-us">
<title>Liftoff News</title>	<title type="text">Liftoff News</title>
<link>http://liftoff.msfc.nasa.gov/</link>	<link rel="alternate" href="http://liftoff.msfc.nasa.gov/" />
<description>Liftoff to Space Exploration.</description>	<subtitle type="text">Liftoff to Space Exploration.</subtitle>
<language>en-us</language>	//see the attribute "xml:lang" of "feed" element
<copyright>Copyright 2002, Spartanburg Herald-Journal</copyright>	<rights>Copyright 2002, Spartanburg Herald-Journal</rights>
<managingEditor>editor@example.com</managingEditor>	<author> <name>editor</name> <email>editor@example.com</email> </author>
<lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>	<updated>2003-06-10T09:41:01Z</updated>
<category domain="http://www.fool.com/cusips">MSFT</category>	<category scheme="http://www.fool.com/cusips" label="MSFT"/>

<category domain="http://www.fool.com/musips">MOTO</category>	<category scheme="http://www.fool.com/musips" label="MOTO"/>
<generator>Weblog Editor 2.0</generator>	<generator>Weblog Editor 2.0</generator>
<image> <url>http://liftoff.msfc.nasa.gov/news.gif</url> <title>Litoff News</title> <link>http://liftoff.msfc.nasa.gov/</link> <width>100</width> <height>100<height> <description>News</description> </image>	<logo>http://liftoff.msfc.nasa.gov/news.gif</logo>
<item>...</item>	<entry>...</entry>
<item>...</item>	<entry>...</entry>
<item>	<entry>
<title>Star City</title>	<title type="text">Star City</title>
<link>http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp</link>	<link rel="alternate" href="http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp"/>
<description>How do Americans get ready to work with Russians aboard the International Space Station? They take a crash course in culture, language and protocol at Russia's Star City.</description>	<summary type="text">How do Americans get ready to work with Russians aboard the International Space Station? They take a crash course in culture, language and protocol at Russia's Star City.</summary>
<author>author1@rssboard.org</author>	<author> <name>author1</name> <email>author1@rssboard.org</email> </author>
<category domain="http://www.fool.com/cusips">MSFT</category> <category domain="http://www.fool.com/musips">MOTO</category>	<category scheme="http://www.fool.com/cusips" label="MSFT" /> <category scheme="http://www.fool.com/musips" label="MOTO" />
<enclosure url="http://www.scripting.com/mp3s/weatherReportSuite.mp3" length="12216320" type="audio/mpeg" />	<link rel="enclosure" type="audio/mpeg" length="12216320" href="http://www.scripting.com/mp3s/weatherReportSuite.mp3"/>
<guid>http://liftoff.msfc.nasa.gov/2003/06/03.html#item573</guid>	<id>http://liftoff.msfc.nasa.gov/2003/06/03.html#item573</id>
<pubDate>Tue, 03 Jun 2003 09:39:21 GMT</pubDate>	<published>2003-06-03T09:39:21Z</published>
</item>	</entry>
</channel> </rss>	</feed>

Examples

The following code examples demonstrate how to convert an RssFeed object into a SyndFeed object and vice versa.

Converting RssFeed object into SyndFeed object:

The following code example demonstrates the conversion of an RssFeed object into a SyndFeed object:

```
@Consumes(MediaType.APPLICATION_XML)
public void readFeed(RssFeed rssFeed) {
    // Map RSS into SyndFeed
    SyndFeed syndFeed = new SyndFeed();
    syndFeed = rssFeed.toSynd(syndFeed);

    // Now access RSS using SyndFeed APIs
    if (syndFeed.getTitle() != null) {
        System.out.println("Title = " + syndFeed.getTitle().getValue());
    }
    if (syndFeed.getSubtitle() != null) {
        System.out.println("Descritpion = " + syndFeed.getSubtitle().getValue());
    }
    ...

    return;
}
```

Converting SyndFeed object into RssFeed object:

The following code example demonstrates the conversion of a SyndFeed object into an RssFeed object:

```
@GET
@Produces(MediaType.APPLICATION_XML)
public RssFeed getFeed() {
    SyndFeed syndFeed = new SyndFeed();
    syndFeed.setTitle(new SyndText("Liftoff News"));
    SyndLink syndLink = new SyndLink();
    syndLink.setRel("alternate");
    syndLink.setHref("http://liftoff.msfc.nasa.gov");
    syndFeed.addLink(syndLink);
    syndFeed.setSubtitle(new SyndText("Liftoff to Space Exploration."));

    SyndEntry syndEntry = new SyndEntry();
    syndEntry.setTitle(new SyndText("Star City"));
    SyndLink syndLink2 = new SyndLink();
    syndLink2.setRel("alternate");
    syndLink2.setHref("http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp");
    syndEntry.addLink(syndLink2);
    syndEntry.setSummary(new SyndText("How do Americans get ready to work with Russians aboard the
International Space Station?"));
    syndFeed.addEntry(syndEntry);

    ...

    //Map SyndFeed into RssFeed
    RssFeed rssFeed = new RssFeed(syndFeed);
    return rssFeed;
}
```

For a further example, please see **ReadRssAsSyndFeed.java** under **examples\client\ReadRSS-client**.

7.5 HTML

This page last changed on Oct 14, 2009 by michael.

HTML

Apache Wink provides a set of providers that are capable of serializing a number of data models (SyndEntry and SyndFeed) as HTML.

	Supported	Media Types	Data model	Provider registration
Read	No	N/A	N/A	N/A
Write	Yes	text/html	org.apache.wink .common.model .synd.SyndFeed org.apache.wink .common.model .synd.SyndEntry	See below.

Activating the HTML provider

The Apache Wink HTML providers are not enabled by default. In order to activate them they must be registered by the **HtmlSyndEntryProvider** and the **HtmlSyndFeedProvider** providers, located in the **org.apache.wink.server.internal.providers.entity.html** package.

As the HTML providers use a jsps in order to generate the representation the HtmlDefaultRepresentation folder must include all its content and subfolders available for the servlet container.

Example

Detailed example of the HTML implementation can be seen at the QADefect example.

7.6 CSV

This page last changed on Oct 14, 2009 by michael.

Comma Separated Values (CSV) Providers

Apache Wink provides a CSV data model and providers for producing and consuming CSV (text/csv). The model is based on a Serialization and a Deserialization interface, in addition to a simple CSV Table class. All of the model classes are located under the org.apache.wink.common.model.csv package.

The following tables list the providers that provide this functionality.

	Supported	Media Types	Data Model	Provider registration
Read	Yes	text/csv	org.apache.wink .common.model .csv.CsvDeserializer org.apache.wink .common.model .csv.CsvTable org.apache.wink .common.model .csv.MultiCsvTable	Not required. Registered by default
Write	Yes	text/csv	org.apache.wink .common.model .csv.CsvSerializer org.apache.wink .common.model .csv.CsvTable org.apache.wink .common.model .csv.MultiCsvTable	Not required. Registered by default

Examples

The following code example demonstrates the reading and writing of CSV documents.

Producing CSV

The following code example demonstrates the creation of a CSV document.

```
@GET
@Produces("text/csv")
public CsvTable getJohns() {
    CsvTable cvs = new CsvTable("Id", "First Name", "Last Name", "Email");
    cvs.addRow("1","John","Kennedy","john@Kennedy.org");
    cvs.addRow("2","John","Lennon","john@Lennon.org");
    cvs.addRow("3","John","Malkovich","john@malkovich.org");
    cvs.addRow("4","John","McCain","john@McCain.org");
    return cvs;
}
```

Consuming CSV

The following code example demonstrates the consumption of a CSV document.

```
@POST
@Consumes("text/csv")
public void postCsv(CsvTable csv) {
   for (String[] row : csv.getRows()) {
      for(String cell: row){
         System.out.print(cell + " ,");
      }
      System.out.print("\n");
   }
}
```

7.7 OpenSearch

This page last changed on Oct 14, 2009 by michael.

OpenSearch Overview

OpenSearch is a collection of simple formats for the sharing of search results.
The OpenSearch description document format is used to describe a search engine that can be used by search client applications.

The OpenSearch response elements can be used to extend existing syndication formats, such as RSS and Atom, with the extra metadata needed to return search results. The OpenSearch document is associated with the "application/opensearchdescription+xml" mime type.

> **Note**
> For more detailed information about Open Search, refers to the Open Search home page at http://www.opensearch.org/Home

Data Model

Apache Wink provides an Open Search data model for producing Open Search Documents. of the model classes are located under the org.apache.wink.common.model.opensearch package. OpenSearchDescription class is used by an application to build Open Search Description documents.

OpenSearch Support

The following table shows the OpenSearch data model and representation in which it can be serialized and de-serialized.

	Supported	Media Types	Data Model	Provider registration
Read	Yes	NA	NA	NA
Write	Yes	application/ opensearchdescription +xml	org.apache.wink .common.model .opensearch .OpenSearchDescription	Not required. Registered by default

Producing an OpenSearch Document Example

The following example demonstrates the creation of an OpenSearch document.

```
    @GET
   @Produces(MediaTypeUtils.OPENSEARCH)
   public OpenSearchDescription getOpenSearch(@Context UriInfo info) {
      String baseUri = info.getAbsolutePath().toString();
      OpenSearchDescription openSearchDescription = new OpenSearchDescription();
      openSearchDescription.setShortName("Example search engine");
      ...
      return openSearchDescription;
}
```

7.8 MultiPart

This page last changed on Oct 14, 2009 by michael.

MultiPart

Apache Wink provides a MultiPart data model and providers for producing and consuming multipart messages (multipart/*). All of the model classes are located under the org.apache.wink.common.model.multipart package distributed with the wink-common jar.

The data model can be used with the wink-server module or with the wink-client module.

	Supported	Media Types	Data Model	Provider registration
Read	Yes	multipart/*	org.apache.wink.common.model.multipart.InMultiPart org.apache.wink.common.model.multipart.BufferedInMultiPart	Not required. Registered by default
Write	Yes	multipart/*	org.apache.wink.common.model.multipart.OutMultiPart org.apache.wink.common.model.multipart.BufferedOutMultiPart	Not required. Registered by default

Serialization and De-serialization

The serialization and de-serialization of a multipart message is performed by the multipart providers. The serialization and de-serialization of the parts that make up the multipart message is performed as if each part is a separate message and in accordance with the JAX-RS specification. This means that every part is serialized and de-serialized using the appropriate provider that matches the binding class and content media type of that specific part.

Main Classes

The multipart data model is comprised of the following main classes:

- **InMultiPart** - is used for de-serialization of an incoming multipart message.
- **InPart** - represents a single part contained in an incoming multipart message.
- **OutMultiPart** - is used for serialization of an outgoing multipart message.
- **OutPart** - represents a single part contained in an outgoing multipart message.

Streaming Multipart

The base multipart classes are designed to handle multipart messages without buffering the data in order to avoid possible memory issues. This means that the data is accessible only once by the use of an iterator.

Buffering Multipart

The BufferedInMultiPart and BufferedOutMultiPart classes are used to handle multipart messages where the complete message is buffered in the memory, allowing random and multiple access of the data. These classes are suitable for situations where the multipart message is small.

Examples

The following examples illustrate the usage of the multipart data model.

Multipart Consumption

The following example illustrates the usage of a streaming multipart message.

```
@Path("files")
@POST
@Produces( MediaType.TEXT_PLAIN)
@Consumes( MediaTypeUtils.MULTIPART_FORM_DATA)
public String uploadFiles(InMultiPart inMP) throws IOException {
   while (inMP.hasNext()) {
      InPart part = inMP.next();
      MultivaluedMap<String, String> heades = part.getHeaders();
      String CDHeader = heades.getFirst("Content-Disposition");
      InputStream is = part.getBody(InputStream.class, null);
      // use the input stream to read the part body
   }
}
```

* Detailed example of the MultiPart implementation can be seen at the MultiPart example.

Buffered Multipart Consumption

The following example illustrates the usage of a buffering multipart message.

```
@Path("users")
@POST
@Consumes( {"multipart/mixed"})
public BufferedOutMultiPart addUsers(BufferedInMultiPart inMP) throws IOException {
   List<InPart> parts = inMP.getParts();
   for (InPart p : parts) {
      User u = p.getBody(User.class, null);
      // use the user object retrieved from the part body

   }
}
```

* Detailed example of the MultiPart implementation can be seen at the MultiPart example.

Appendix A - Feeds Support

This page last changed on Oct 15, 2009 by shivakumar.

Migration from Apache Abdera to Apache Wink

Apache Wink is an excellent solution for consuming and producing Atom, APP and RSS documents. The following section describes how to migrate from Apache Abdera to Apache Wink by providing a set of examples that cover most use cases.

Advantages of Apache Wink over Apache Abdera

- Standardized APIs (using JAX-RS and JAXB)
- Support for handling XML and JSON more easily
- Support for handling RSS and ATOM more easily

This section contains the following topics:

---

1) Consuming Atom Documents

The following code example demonstrates the consumption of Atom documents using Apache Abdera.

Apache Abdera - Click on link to Download - ConsumeAtomUsingAbdera.java

```
Abdera abdera = new Abdera();
Parser parser = abdera.getParser();
URL url = new URL("http://alexharden.org/blog/atom.xml");
Document<Feed> doc = parser.parse(url.openStream());
Feed feed = doc.getRoot();
System.out.println(feed.getTitle());
for (Entry entry : feed.getEntries()) {
   System.out.println("\t" + entry.getTitle());
}
```

The following code example demonstrates the consumption of Atom documents using Apache Wink.

Apache Wink - Click on link to Download - ConsumeAtomUsingWink.java

```
RestClient client = new RestClient();
Resource resource = client.resource("http://alexharden.org/blog/atom.xml");
AtomFeed feed = resource.accept(MediaType.APPLICATION_ATOM_XML).get(AtomFeed.class);
```

```
System.out.println(feed.getTitle().getValue());
for (AtomEntry entry : feed.getEntries()) {
   System.out.println("\t" + entry.getTitle().getValue());
}
```

2) a) Producing Atom Documents

The following code example demonstrates the production of Atom documents using Apache Abdera.

Apache Abdera - Click on links to Download - ProduceAtomUsingAbdera.java ProduceAtomUsingAbdera_web.xml

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   Abdera abdera = new Abdera();
   Feed feed = abdera.newFeed();

   feed.setId("tag:example.org,2007:/foo");
   feed.setTitle("Test Feed");
   feed.setSubtitle("Feed subtitle");
   feed.setUpdated(new Date());
   feed.addAuthor("Shiva HR");
   feed.addLink("http://example.com");
   feed.addLink("http://example.com/foo", "self");

   Entry entry = feed.addEntry();
   entry.setId("tag:example.org,2007:/foo/entries/1");
   entry.setTitle("Entry title");
   entry.setSummaryAsHtml("<p>This is the entry title</p>");
   entry.setUpdated(new Date());
   entry.setPublished(new Date());
   entry.addLink("http://example.com/foo/entries/1");

   feed.getDocument().writeTo(response.getWriter());
}
```

The following code example demonstrates the production of Atom documents using Apache Wink.

Apache Wink - Click on links to Download - ProduceAtomUsingWink.java ProduceAtomUsingWink_web.xml

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   AtomFeed feed = new AtomFeed();
   feed.setId("tag:example.org,2007:/foo");
   feed.setTitle(new AtomText("Test Feed"));
   feed.setSubtitle(new AtomText("Feed subtitle"));
   feed.setUpdated(new Date());

   AtomPerson person = new AtomPerson();
   person.setName("Shiva HR");
```

```
        feed.getAuthors().add(person);

        AtomLink link1 = new AtomLink();
        link1.setHref("http://example.com");
        feed.getLinks().add(link1);

        AtomLink link2 = new AtomLink();
        link2.setHref("http://example.com/foo");
        link2.setRel("self");
        feed.getLinks().add(link2);

        AtomEntry entry = new AtomEntry();
        entry.setId("tag:example.org,2007:/foo/entries/1");
        entry.setTitle(new AtomText("Entry title"));

        AtomText summary = new AtomText();
        summary.setType(AtomTextType.html);
        summary.setValue("<p>This is the entry title</p>");
        entry.setSummary(summary);

        entry.setUpdated(new Date());
        entry.setPublished(new Date());

        AtomLink link3 = new AtomLink();
        link3.setHref("http://example.com/foo/entries/1");
        entry.getLinks().add(link3);

        feed.getEntries().add(entry);

        AtomFeed.marshal(feed, response.getOutputStream());
    }
```

2) b) Producing Atom Documents - the JAX-RS way

A more elegant way of producing Atom documents using Apache Wink is the JAX-RS way as described below:

1. Open the Eclipse development environment and create a "Dynamic Web Project".
2. Add Apache Wink & its dependent JARs under Java EE Module Dependencies.
3. Create a POJO class and a method that creates Atom feed document. Annotate the class & its methods with the required JAX-RS annotations as below:
   ProduceAtom.java
4. Add org.apache.wink.server.internal.servlet.RestServlet into web.xml and specify the path of above Resource class in it's init-param.
   See ProduceAtomWinkElegant_web.xml and application
5. Deploy the web-application and access it using the url http://localhost:8080/ProduceAtom_Wink_Elegant/rest/getAtom
6. Final WAR -> ProduceAtom_Wink_Elegant.zip (add Wink & its dependent JARs under ProduceAtom_Wink_Elegant\WEB-INF\lib and re-zip it as WAR).

---

3) Consuming RSS Documents

The following code example demonstrates the consuming of RSS documents using Apache Abdera.

Apache Abdera - Click on link to Download - ConsumeRssUsingAbdera.java

```
public static void main(String[] args) throws ParseException, IOException {
    System.out.println("Consuming RSS Documents using Abdera...\n");
    Abdera abdera = new Abdera();
    Parser parser = abdera.getParser();
    URL url = new URL("http://www.rssboard.org/files/sample-rss-2.xml");
    Document<RssFeed> doc = parser.parse(url.openStream());
    RssFeed rssFeed = doc.getRoot();
    System.out.println("Title: " + rssFeed.getTitle());
    System.out.println("Description: " + rssFeed.getSubtitle() + "\n");
    int itemCount = 0;
    for (Entry entry : rssFeed.getEntries()) {
        System.out.println("Item " + ++itemCount + ":");
        System.out.println("\tTitle: " + entry.getTitle());
        System.out.println("\tPublish Date: " + entry.getPublished());
        System.out.println("\tDescription: " + entry.getContent());
    }
}
```

The following code example demonstrates the consuming of RSS documents using Apache Wink.

Apache Wink - Click on link to Download - ConsumeRssUsingWink.java

```
public static void main(String[] args) {
    System.out.println("Consuming RSS Documents using Apache Wink...\n");
    RestClient client = new RestClient();
    String url = "http://www.rssboard.org/files/sample-rss-2.xml";
    Resource resource = client.resource(url);
    RssFeed rss = resource.accept(MediaType.APPLICATION_XML).get(RssFeed.class);
    RssChannel channel = rss.getChannel();
    System.out.println("Title: " + channel.getTitle());
    System.out.println("Description: " + channel.getDescription() + "\n");
    int itemCount = 0;
    for (RssItem item : channel.getItems()) {
        System.out.println("Item " + ++itemCount + ":");
        System.out.println("\tTitle: " + item.getTitle());
        System.out.println("\tPublish Date: " + item.getPubDate());
        System.out.println("\tDescription: " + item.getDescription());
    }
}
```

---

4) Producing RSS Documents

Apache Abdera

Apache Abdera version 0.4 does not support RSS write.

Apache Wink

Same as in 2) b) Producing Atom Documents - the JAX-RS way. However the resource method now returns an RssFeed object instead of AtomFeed object.

Apache Wink - Click on link to Download - ProduceRss_Wink_Elegant.zip

```
@Path("/getRss")
public class ProduceRss {
   @GET
   @Produces(MediaType.APPLICATION_XML)
   public Rss getRss() {
      RssFeed rss = new RssFeed();

      RssChannel channel = new RssChannel();
      channel.setTitle("Liftoff News");
      channel.setLink("http://liftoff.msfc.nasa.gov");
      channel.setDescription("Liftoff to Space Exploration.");
      channel.setPubDate(new Date().toString());

      RssItem item = new RssItem();
      item.setTitle("Star City");
      item.setLink("http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp");
      item.setDescription("How do Americans get ready to work with Russians aboard the International
 Space Station?");

      channel.getItem().add(item);
      rss.setChannel(channel);
      return rss;
   }
}
```

5) Writing Atom Publishing Protocol (APP) Server

The following steps explain how to implement an APP server as described in the following beautiful article by James Snell: http://www.ibm.com/developerworks/library/x-atompp1/

Apache Abdera

1. Open the Eclipse development environment and create a "Dynamic Web Project".
2. Add Apache Abdera & its dependent JARs under Java EE Module Dependencies.
3. Add the following CollectionAdapter and Provider classes under src/myPackage directory:
   APP_CollectionAdapter.java APP_ContentProvider.java
4. Add org.apache.abdera.protocol.server.servlet.AbderaServlet into web.xml and point the following init paramters to the classes added above.
   org.apache.abdera.protocol.server.Provider
   org.apache.abdera.protocol.server.CollectionAdapter
   APP_Server_Abdera_web.xml
5. Add the following index.jsp which has help on how to perform the APP operations:
   APP_Server_Abdera_index.jsp
6. Deploy and run the application.

Final WAR -> APP_Server_Abdera.zip (add Apache Abdera & its dependent JARs under APP_Server_Abdera\WEB-INF\lib and re-zip it as WAR).

Apache Wink

1. Open the Eclipse development environment and create a "Dynamic Web Project".
2. Add Apache Wink & its dependent JARs under Java EE Module Dependencies.
3. Add the following Resource class under src/myPackage directory: EntriesCollection.java

4. Add org.apache.wink.server.internal.servlet.RestServlet into web.xml and specify the path of above Resource class in it's init-param. APP_Server_Wink_web.xml APP_Server_Wink_application
5. Add the following index.jsp which has help on how to perform the APP operations: APP_Server_Wink_index.jsp
6. Deploy and run the application.

Final WAR -> APP_Server_Wink.zip (add Apache Wink & its dependent JARs under APP_Server_Wink \WEB-INF\lib and re-zip it as WAR)

References

- Apache Wink's "SimpleDefects" example: http://svn.apache.org/repos/asf/incubator/wink/tags/wink-0.1-incubating/wink-examples/apps/SimpleDefects/src/main/java/org/apache/wink/example/simpledefects/resources/DefectsResource.java
- Abdera Feed Sample shipped with IBM WebSphere Feature Pack for Web 2.0 http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.ajax.feed.samples.help/docs/GettingStarted_useage.html
- Abdera Server Implementation Guide -> http://cwiki.apache.org/ABDERA/server-implementation-guide.html
- Abdera Collection Adapter Implementation Guide -> http://cwiki.apache.org/ABDERA/collection-adapter-implementation-guide.html

---

6) Writing Atom Publishing Protocol (APP) Client

In order to write an Atom Publishing Protocol client refer to the following examples.

Important Note
Make sure that the APP_Server_Abdera.war and the APP_Server_Wink.war provided in the previous example are deployed before running these examples.

Apache Abdera - Click on link to Download - APP_Client_Abdera.java

1. Acessing Service Document:

```
Document<Service> introspection = abderaClient.get(SERVICE_URL).getDocument();
Service service = introspection.getRoot();
List<Workspace> workspaces = service.getWorkspaces();
for (Workspace workspace : workspaces) {
    System.out.println("\t" + workspace.getTitle());
    List<Collection> collections = workspace.getCollections();
    for (Collection collection : collections) {
        System.out.println("\t" + collection.getTitle() + "\t:\t" + collection.getHref());
    }
    System.out.print("\n");
}
```

2. Getting a Feed

```
RequestOptions opts = new RequestOptions();
opts.setContentType("application/atom+xml;type=feed");
ClientResponse response = abderaClient.get(FEED_URL, opts);
Feed feed = (Feed)response.getDocument().getRoot();
```

3. Posting an entry to a Feed

```
RequestOptions opts = new RequestOptions();
opts.setContentType("application/atom+xml;type=entry");
ClientResponse response = abderaClient.post(FEED_URL, newEntry, opts);
```

4. Putting a change to an Entry

```
RequestOptions opts = new RequestOptions();
opts.setContentType("application/atom+xml;type=entry");
ClientResponse response = abderaClient.put(ENTRY_URL, changedEntry.getDocument(), opts);
```

5. Getting an Entry

```
RequestOptions opts = new RequestOptions();
opts.setContentType("application/atom+xml;type=entry");
ClientResponse response = abderaClient.get(ENTRY_URL, opts);
Entry entry = (Entry)response.getDocument().getRoot();
```

6. Deleting an Entry

```
ClientResponse response = abderaClient.delete(ENTRY_URL);
```

Apache Wink - Click on link to Download - APP_Client_Wink.java

1. Acessing Service Document:

```
Resource resource = restClient.resource(SERVICE_URL);
AppService service = resource.accept(MediaTypeUtils.ATOM_SERVICE_DOCUMENT).get(AppService.class);
List<AppWorkspace> workspaces = service.getWorkspace();
for (AppWorkspace workspace : workspaces) {
    System.out.println("\t" + workspace.getTitle().getValue());
    List<AppCollection> collections = workspace.getCollection();
    for (AppCollection collection : collections) {
        System.out.println("\t" + collection.getTitle().getValue()
            + "\t:\t"
            + collection.getHref());
    }
    System.out.print("\n");
}
```

2. Getting a Feed

```
Resource feedResource = restClient.resource(FEED_URL);
AtomFeed feed = feedResource.accept(MediaType.APPLICATION_ATOM_XML).get(AtomFeed.class);
```

3. Posting an entry to a Feed

```
Resource feedResource = restClient.resource(FEED_URL);
ClientResponse response =
   feedResource.contentType(MediaType.APPLICATION_ATOM_XML).post(newEntry);
```

4. Putting a change to an Entry

```
Resource feedResource = restClient.resource(ENTRY_URL);
ClientResponse response =
   feedResource.contentType(MediaType.APPLICATION_ATOM_XML).put(changedEntry);
```

5. Getting an Entry

```
Resource feedResource = restClient.resource(ENTRY_URL);
AtomEntry atomEntry = feedResource.accept(MediaType.APPLICATION_ATOM_XML).get(AtomEntry.class);
```

6. Deleting an Entry

```
Resource feedResource = restClient.resource(ENTRY_URL);
ClientResponse response = feedResource.delete();
```

Appendix B - Google App Engine

This page last changed on Feb 16, 2010 by bluk.

Google App Engine

Apache Wink can be run using Google App Engine as the core functionality is fully compatible. However, due to some of the App Engine limitations, a number of the additional functions may not work correctly.

Logging

Google uses java.util.logging, also referred to as the JDK 1.4 logging. Apache Wink uses slf4j. In order to enable Wink's logging function, replace the `slf4j-simple-<version>.jar` with the `slf4j-jdk14-<version>.jar`.

In order to view all of Wink's messages place the following property in the logging.properties file:

```
org.apache.wink.level=ALL
```

Everything that the servlet writes to the standard output stream (System.out) and standard error stream (System.err) is captured by the App Engine and then recorded in the application logs.

Useful Tip
In order to gain an in depth understanding refer to Logging.

When running with Spring, make sure that you have `jcl-over-slf4j-<version>.jar` in the classpath. This jar is needed, since Spring uses commons-logging.

Additional Issues

The following section contains "**additional Issues**" that maybe experience while trying to run custom developer code in the Google App Engine. It is advantageous to read the relevant documentation as it provides important information, that will save valuable developer time that is required to ascertain why the application behaves differently in comparison to a regular servlet container.

Context Path

Usually the URI in a servlet container looks like this:

```
requestURI = contextPath + servletPath + pathInfo
```

while in most servlet containers context path is a war name (or it can be configured), in App Engine it's just empty.
So if your servlet is mapped to "rest", access

```
http://host:port/rest
```

.

References

- App Engine Java Overview
- Will it play in App Engine