HP Symphony 2.0 Feature Set

Software Version: 2.0

[The Symphony 2.0 Feature Set internal draft document is a broad scope document that provides detailed information about Symphony 2.0 strategy and design].

Table of Contents

List of Tables

1. Introduction

The purpose of this document is to provide detailed information about Symphony 2.0 and describe the additional features that the Symphony 2.0 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 Symphony 2.0 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 Symphony.

1.1.1. Important Note

This Features Set Document is a Preliminary Draft

This document contains technical specification materials compiled by the Symphony development team in draft format.

1.2. Target Audience

In order to understand the contents of this document the reader is required to have read the JAX-RS specification and have a rudimentary understanding of the specification and the terminology used to describe the feature set.

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

1.3. JAX-RS Compliancy

Symphony 2.0 is a complete implementation of the JAX-RS v1.0 specification.

The JAX-RS TCK tests still need to performed in order to be able to declare that it is JAX-RS compliant.

2. Symphony 2.0 Architecture

The following chapter describes the basic concepts and building blocks of Symphony 2.0 and explains the high-level architecture of the Symphony runtime.

2.1. Symphony Runtime 2.0 Architecture Overview

The Symphony 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 Symphony engine for processing.

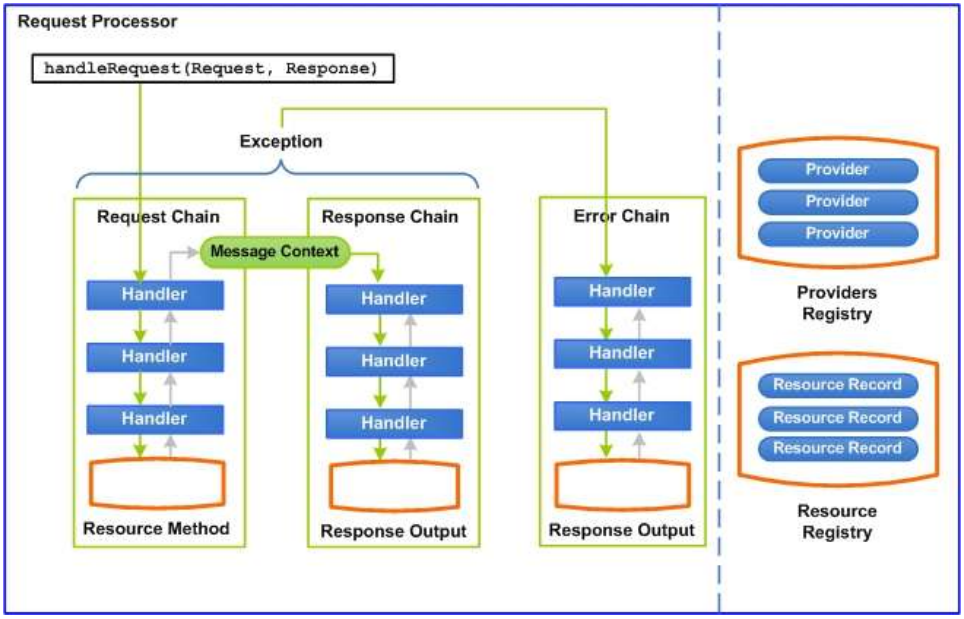

Figure 1: Request Processor Architecture

The above diagram illustrates the core components of the Symphony runtime. The Symphony 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 for generating a response.

In case of an error, the RequestProcessor invokes the Error chain with the generated exception for producing the appropriate response.

The Symphony runtime maintains providers and resources in two registries, the **Providers Registry** and the **Resource Registry** utilizing them during request processing.

2.2. Request Processor

The **RequestProcessor** is the Symphony 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 Symphony runtime to handle the request. The RequestProcessor first runs the **Request Handler Chain** to invoke the resource method and then the **Response Handler Chain** to produce the response.

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

2.3. Deployment Configuration

The Symphony runtime is initialized with an instance of a **DeploymentConfiguration**. The Deployment Configuration holds the runtime configuration, including the handler chains, registries and configuration properties.

The Deployment Configuration is initialized with an instance of a JAX-RS Application used for obtaining user resources and providers.

2.3.1. Customization

The Deployment Configuration is customized by extending the DeplymentConfiguration class, overriding specific methods and specifying the new class in the web.xml file of the application.

In order to specify a different Deployment Configuration class instead of the default Deployment Configuration, the value of the **symphony.deploymentConfiguration** init parameter must be set to be the fully qualified name of the customized configuration class.

```
<servlet>
  <servlet-name>restSdkService</servlet-name>
  <servlet-class>
    com.hp.symphony.server.internal.servlet.RestServlet
  </servlet-class>
  <init-param>
    <param-name>symphony.deploymentConfiguration</param-name>
    <param-value>com.hp.example.MyDeploymentConfig</param-value>
  </init-param>
</servlet>
```

The following table details the customizable methods of the Deployment Configuration class.

Deployment Configuration

Table 1: Deployment Configuration Customizable Methods

Method	Description
initAlternateShortcutMap	Initializes the AlternateShortcutMap. Refer to section 3.8
initMediaTypeMapper	Initializes the MediaTypeMapper. Refer to section 3.7
initRequestUserHandlers	Return a list of User Handler instances to embed in the Request chain. Refer to section 13.3
initResponseUserHandlers	Return a list of User Handler instances to embed in the Response chain. Refer to section 13.4
initErrorUserHandlers	Return a list of User Handler instances to embed in the Error chain. Refer to section 13.5

2.4. Handler Chains

The handler chain pattern is used by the Symphony runtime for implementing the core functionalities.

There are three handler chains utilized by the Symphony runtime:

- **RequestHandlersChain**
- **ResponseHandlersChain**
- **ErrorHandlersChain**

Refer to chapter 13 for more information on **Handler Chains.**

2.5. Registries

The Symphony 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, Symphony supports the prioritization of resources and providers.

Refer to chapter 3, section 3.4 for more information on **Priorities.**

2.5.1. Resources Registry

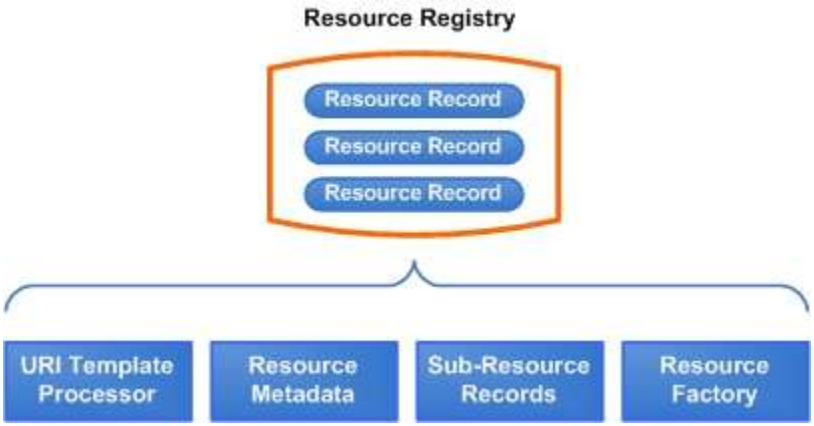

Figure 2: Resource Registry

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

2.5.2. Providers Registry

The providers registry maintains of all of the system and user providers and manages them in an efficient way.

3. Registration and Configuration

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

3.1. Simple Application

Symphony provides the **SimpleSymphonyApplication** 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
```

3.1.1. Specifying the Simple Application File Location

The path to a simple application file is configured via the **symphony.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>
    com.hp.symphony.server.internal.servlet.RestServlet
  </servlet-class>
  <init-param>
    <param-name>symphony.applicationConfigLocation</param-name>
    <param-value>/WEB-INF/providers;/WEB-INF/resources</param-value>
  </init-param>
</servlet>
```

3.2. Symphony Application

Symphony extends the javax.ws.rs.core.Application class with the com.hp.symphony.common.SymphonyApplication class in order to provide the Dynamic Resources and the Priorities functionality.

> Refer to chapter 3, sections 3.3 and 3.4 for more information on **Dynamic Resources and Priorities.**

An application may provide an instance of SymphonyApplication to the Symphony runtime as specified by the JAX-RS specification.

3.3. 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 com.hp.symphony.server.DynamicResource interface and must not be annotated with the @Path annotation.

3.3.1. 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;
    }
}
```

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.

3.3.2. Usage

A Dynamic Resource is a resource class that implements the com.hp.symphony.server.DynamicResource interface or extends the com.hp.symphony.server.AbstractDynamicResource convenience class.

A Dynamic Resource is not registered in Symphony 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 SymphonyApplication#getInstances()method must be used.

> Refer to chapter 3, section 3.2 for more information about **Symphony Application**.

3.3.3. 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 11 for more information about **Spring Integration**.

3.4. Priorities

Although JAX-RS defines the algorithm for searching for resources and providers, Symphony extends this algorithm by providing the ability to specify priorities on them. 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 SymphonyApplication 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.

3.4.1. Resource Priorities

Priorities on resources are useful in 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 Symphony 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.

3.4.2. Provider Priorities

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

The Symphony 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.

3.5. Properties

Symphony provides a properties file in order to enable simple customizations. By default, Symphony predefines default values for all possible properties.

Customization Properties

Table 2: Symphony Customization Properties

Property Name	Description	Default Value	Ref
symphony.http.uri	URI that is used by the Link Builders in case of HTTP	Use the URI from the request.	chapter 4
symphony.https.uri	URI used by the Link Builders in case of HTTPS.	Use the URI from the request.	chapter 4
symphony.context.uri	Context path used by the Link Builders.	Use the context path from the request.	chapter 4
symphony.defaultUrisRelative	Indicates if URIs generated by the Link Builders are absolute or relative. Valid values: true or false	**true** – links will be relative.	chapter 4

symphony.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	chapter 4
symphony.searchPolicyContinuedSearch	Indicates if continues search is enabled. Valid values: true, false	**false** – continued search is disabled.	chapter 8
symphony.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	chapter 10
symphony.serviceDocumentCssPath	Defines path to a css file that is used in the HTML Service Document generation. Relevant only if HTML Service Document is defined.	No css file defined.	chapter 10

3.5.1. Custom Properties File Definition

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

```
<servlet>
  <servlet-name>restSdkService</servlet-name>
  <servlet-class>
    com.hp.symphony.server.internal.servlet.RestServlet
  </servlet-class>
  <init-param>
    <param-name>symphony.propertiesLocation</param-name>
    <param-value>/WEB-INF/configuration.properties</param-value>
  </init-param>
  <init-param>
    <param-name>symphonyApplicationConfigLocation</param-name>
    <param-value>/WEB-INF/application</param-value>
  </init-param>
  <load-on-startup>0</load-on-startup>
</servlet>
```

3.6. Runtime Registration

Symphony provides several APIs for Runtime Registration. The APIs appear in the com.hp.symphony.server.utils.RegistrationUtils class.

The most important method is the one that registers an instance of the javax.ws.rs.core.Application class

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

Note

Double Registration

Registration is ignored and a warning is printed to the log if the same instance is registered more than once.

3.7. 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.

Symphony 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.

Mappings

Table 3: Predefined Mappings

User Agent	Content-Type	Map To
Mozilla/	application/atom+xml	text/xml
Mozilla/	application/atomsvc+xml	text/xml
Mozilla/	application/opensearchdescription+xml	text/xml

3.7.1. Customizing Mappings

In order to customize these mappings the application should create an instance of a com.hp.symphony.server.internal.MediaTypeMapper class and set it on the DeploymentConfiguration instance.

Refer to chapter 2, section 2.3.1 for more information **on Customizing the Default Deployment Configuration.**

3.8. Alternative Shortcuts

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

A request to "/entry?alt=application/xml" specifies that the requested response media type is application/xml.

Symphony 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.

Shortcuts

Table 4: 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

3.8.1. Customizing Shortcuts

The shortcuts table can be customized by overriding the DeploymentConfiguration class.

Refer to chapter 2, section 2.3 for more information about **Deployment Configuration**.

4. 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 according to the invoked method type. The link builder interfaces reside in the **com.hp.symphony.server.utils** package.

4.1. 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.

Symphony 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.

Symphony 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.

4.2. The "alt" Query Parameter

Symphony 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 **symphony.addAltParam** key of the configuration properties file or by calling the LinksBuilder#addAltParam() method.

Refer to chapter 3, section 3.5 for more information on **Configuration Properties**.

4.3. 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.

4.3.1. 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();
    List<SyndLink> systemLinks = builder.build(null);
    ...
  }
}
```

The DefectResource#getXml() method is invoked when a GET request for application/xml is made to /defects/3. The Symphony 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"/>`

4.4. 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.

4.5. 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 **symphony.defaultUrisRelative** key set in the application configuration properties file. The value must be either true or false.

Refer to chapter 3, section 3.5 for more information on the **Configuration Properties file.**

4. If the configuration key does not exist, then use true.

5. 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 Symphony runtime to retrieve the actual response entity by invoking the appropriate **entity-producing method** of the asset.

Refer to chapter 5, section 5.3.1 for more information on **Entity-Producing Methods**.

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

Refer to chapter 5, section 5.3.2 for more information on **Entity-Consuming Methods**.

5.1. 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.

Note

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.

5.2. Lifecycle

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

5.2.1. Response Entity Asset

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

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

5.2.2. Request Entity Asset

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

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

5.3. 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.

5.3.1. 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 Symphony runtime will not invoke an entity-producing method whose effective value of @Produces does not match the request Accept header

5.3.2. 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 Symphony runtime will not invoke an entity-consuming method whose effective value of @Consumes does not match the request Content-Type header.

5.4. Parameters

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

5.5. Return Type

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

5.6. Exceptions

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

5.7. 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.

5.8. 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.

Refer to chapter 3, section 6.3.5 for more information on **Asset Providers**.

5.8.1. 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.

- Collect all the entity-consuming methods of the asset. These are the public methods annotated with @Consumes annotation.
- 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/* > */*.
- Select the first method that supports the media type of the request entity body as provided to the AssetProvider#isReadable() method, and return true.
- If no entity-consuming method supports the media type of the request entity body, return false. The Symphony runtime continues searching for a different provider to handle the asset as a regular entity.

5.8.2. 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.

- Collect all the entity-producing methods of the asset. These are the public methods annotated with @Produces annotation.
- 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/* > */*.
- Select the first method that supports the media type of the response entity body as provided to the AssetProvider#isWriteable()method and return true.
- If no entity-producing method supports the media type of the response entity body, return false. The Symphony runtime continues searching for a different provider to handle the asset as a regular entity.

5.9. Asset Example

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

The DefectAsset class is the asset backed by an instance of a “Defect” bean. The DefectResource class is a resource that is anchored to the URI path “defects/{id}” within the Symphony runtime.

DefectAsset Class

```
@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

```
@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

- A client issues an HTTP GET request with a URI=”/defects/1” and Accept Header= “application/xml”
- The Symphony runtime analyzes the request and invokes the DefectResource#getDefect() resource method.
- The DefectResource#getDefect() resource method creates an instance of DefectAsset and populates it with defect “1” data.
- The `DefectResource#getDefect()` resource method returns the DefectAsset instance back to Symphony runtime.
- The Symphony runtime analyzes the asset and invokes the DefectAsset#getDefect() entity-producing method to obtain the reference to the “Defect” bean.
- The “Defect” bean is serialized by Symphony runtime as an XML using the appropriate provider.

Scenario Explanation 2

- A Client issues an HTTP GET request with a URI=”/defects/1” and Accept Header= “text/html”
- The Symphony runtime analyzes the request and invokes the DefectResource#getDefect() resource method

- The DefectResource#getDefect() resource method creates an instance of DefectAsset and populates it with defect “1” data.
- The DefectResource#getDefect() method returns the populated asset back to the Symphony runtime.
- The Symphony runtime analyzes the asset and invokes the DefectAsset#getDefectAsHtml() entity-producing method in order to obtain the reference to the “Defect” bean.
- The “Defect” is serialized by Symphony runtime as an HTML using the appropriate provider.

Scenario Explanation 3

- A Client issues an HTTP PUT request with a URI=”/defects/1” and Accept Header= “text/html”
- The Symphony 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 Symphony runtime
  - The DefectAsset#setDefect() entity-consuming method is invoked in order to populate the DefectAsset with the defect data.

6. Providers

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

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

6.1. Scoping

The JAX-RS specification defines that by default, a singleton instance of each provider class is instantiated for each JAX-RS application. Symphony 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 (section0) 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.

6.1.1. 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>{
    ...
}
```

6.1.2. Singleton Example 1

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

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

6.1.3. 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>{
    ...
}
```

6.2. Priority

Symphony provides a method for setting a priority for a provider.

Refer to chapter 3, section 3.4.2 for more information on **Provider Priorities.**

6.3. Out-of-the-Box Implementations

The following section describes the Symphony providers that are an addition to the JAX-RS requirements.

6.3.1. Atom Providers

Symphony provides a set of entity providers that are capable of mapping Atom Feed and Atom Entry XML documents to and from an Atom data model.

> Refer to chapter 9 for more information on **Data Models**.

The following tables list these providers.

AtomFeedProvider

Table 5: AtomFeedProvider

	Supported	Media Types	Entity
Read	Yes	application/atom+xml	AtomFeed
Write	Yes	application/atom+xml	

AtomFeedSyndFeedProvider

Table 6: AtomFeedSyndFeedProvider

	Supported	Media Types	Entity
Read	Yes	application/atom+xml	SyndFeed
Write	Yes	application/atom+xml	

AtomFeedJAXBElementProvider

Table 7: AtomFeedJAXBElementProvider

	Supported	Media Types	Entity
Read	Yes	application/atom+xml	JAXBElement<AtomFeed>
Write	Yes	application/atom+xml	

AtomEntryProvider

Table 8: AtomEntryProvider

	Supported	Media Types	Entity
Read	Yes	application/atom+xml	AtomEntry
Write	Yes	application/atom+xml	

AtomEntrySyndEntryProvider

Table 9: AtomEntrySyndEntryProvider

	Supported	Media Types	Entity
Read	Yes	application/atom+xml	SyndEntry
Write	Yes	application/atom+xml	

AtomEntryJAXBElementProvider

Table 10: AtomEntryJAXBElementProvider

	Supported	Media Types	Entity
Read	Yes	application/atom+xml	JAXBElement<AtomEntry>
Write	Yes	application/atom+xml	

6.3.2. APP Providers

Symphony provides a set of providers that are capable of mapping APP Service Document and APP Categories data models to their xml representations. The following tables list these providers.

> Refer to chapter 9, section 9.5 in chapter 9 for more information on the **Atom Publishing Protocol.**

AppServiceProvider

Table 11: AppServiceProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/atomsvc+xml	AppService

AppCategoriesProvider

Table 12: AppCategoriesProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/atomcat+xml	AppCategories

CategoriesProvider

Table 13: CategoriesProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/atomcat+xml	Categories

6.3.3. OpenSearch Provider

Symphony provides a single provider that is capable of serializing the OpenSearch data model.

> Refer to the xml representations chapter 9, section 9.7 for more information on **OpenSearch**.

OpenSearchDescriptionProvider

Table 14: OpenSearchDescriptionProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/opensearchdescription+xml	OpenSearchDescription

6.3.4. Json Providers

Symphony provides a set providers that are capable of serializing a number of data models (JSONObject, JAXBElement, SyndEntry, SyndFeed) into JSON representations. The following tables list these providers.

JsonProvider

Table 15: JsonProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/json , application/javascript	JSONObject

JsonJAXBProvider

Table 16: JsonJAXBProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/json , application/javascript	JAXB object, JAXBElement<?>

JsonSyndEntryProvider

Table 17: JsonSyndEntryProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/json , application/javascript	SyndEntry

JsonSyndFeedProvider

Table 18: JsonSyndFeedProvider

	Supported	Media Types	Entity
Read	No	N/A	N/A
Write	Yes	application/json , application/javascript	SyndFeed

6.3.5. Asset Provider

Symphony provides a special provider that is responsible for reading and writing Asset objects.

> Refer to chapter 5 for more information on **Assets**.

AssetProvider

Table 19: AssetProvider

	Supported	Media Types	Entity
Read	Yes	*/*	POJOs annotated with **@Asset** annotation.
Write	Yes	*/*	POJOs annotated with **@Asset** annotation.

6.3.6. HTML Providers

Symphony provides a set of providers that are capable of serializing a number of data models (SyndEntry, SyndFeed and HtmlDescriptor) as HTML. The following tables list these providers.

HtmlProvider

Table 20: HtmlProvider

	Supported	Media Types	Entity
Read	NO	N/A	N/A
Write	Yes	text/html	HtmlDescriptor

HtmlSyndEntryProvider

Table 21: HtmlSyndEntryProvider

	Supported	Media Types	Entity
Read	NO	N/A	N/A
Write	Yes	text/html	SyndEntry

HtmlSyndFeedProvid

Table 22: HtmlSyndFeedProvid

	Supported	Media Types	Entity
Read	NO	N/A	N/A
Write	Yes	text/html	SyndFeed

6.3.7. CSV Providers

Symphony supports the serializing and de-serializing of data as a CSV.

> Refer to chapter 9, section 9.6 for more information on **Comma Separated Values.**

The following tables list the providers that provide this functionality.

CsvSerializerProvider

Table 23: CsvSerializerProvider

	Supported	Media Types	Entity
Read	NO	N/A	N/A
Write	Yes	text/csv	CsvSerializer

CsvDeserializerProvider

Table 24: CsvDeserializerProvider

	Supported	Media Types	Entity
Read	Yes	text/csv	CsvDeserializer
Write	NO	N/A	N/A

7. Annotations

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

7.1. @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.

> Refer to chapter 10 for more information on **APP Service Document**.

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

Annotation Specification

Table 25: @Workspace Annotation Specification

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

7.1.1. @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>
```

7.2. @Asset Annotation

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

> Refer to chapter 5 for more information on **Assets**.

Annotation Specification

Table 26: @Asset Annotation Specification

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

7.3. @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.

Symphony provides the **@Scope** annotation to specify the lifecycle of a provider or resource.

Annotation Specification

Table 27: @Scope Annotation Specification

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

7.3.1. Resource Example

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

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

7.3.2. 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> {
    ...
}
```

7.4. @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 Symphony 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.

Annotation Specification

Table 28: @Parent Annotation Specification

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

Example

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

```
@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”.

8. Resource Matching - Continued Search

Symphony 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.

8.1. 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, Symphony 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 Symphony 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.

8.2. Configuration

The Continued Search mode is activated by setting the value of the **symphony.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.

9. Data Models

The following chapter describes the out-of-the-box data models provided and supported by the Symphony runtime.

9.1. JAXB

Symphony supports JAXB objects for consuming and producing XML (application/xml), and for producing JSON (application/json).

An application may provide a ContextResolver for obtaining the JAXBContext of the JAXB object. If no suitable JAXBContext is obtained from a ContextResolver, then the Symphony runtime uses a default JAXBContext initialized with the JAXB object.

9.2. JSON

Symphony provides a JSON data model for producing JSON (application/json). All of the model classes are located under the **com.hp.symphony.common.model.json** package.

9.3. Syndication

Symphony provides a syndication data model for producing Atom (application/atom+xml), HTML (text/html), JSON (application/json) and CSV (text/csv), and for consuming Atom and CSV. All of the model classes are located under the **com.hp.symphony.common.model.synd** package.

9.4. Atom

Symphony provides an Atom data model for consuming and producing Atom feeds and entries (application/atom+xml). All of the model classes are located under the **com.hp.symphony.common.model.atom** package.

9.5. Atom Publishing Protocol (APP)

Symphony provides an Atom Publishing Protocol data model for producing Service Documents (application/atomsvc+xml) and Categories Documents (application/atomcat+xml). The APP data model can also be used to produce Service and Categories documents in HTML (text/html) and JSON (application/json) formats. All of the model classes are located under the **com.hp.symphony.common.model.app** package.

9.6. Comma Separated Values (CSV)

Symphony provides a CSV data model 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 **com.hp.symphony.common.model.csv** package.

9.7. OpenSearch

Symphony provides an OpenSearch data model for producing OpenSearch Description Documents (application/opensearchdescription+xml). All of the model classes are located under the **com.hp.symphony.common.model.opensearch** package.

10. APP Service Document

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

10.1. Enabling the APP Service Document Auto Generation

APP Service Document generation is activated by setting the symphony.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”.

10.2. Adding Resources to APP Service Document

Symphony 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.

Refer to chapter 7, section 7.1 for more information on the **@Workspace annotation**.

10.2.1. 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>
```

10.3. APP Service Document HTML Styling

Symphony 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 **symphony.serviceDocumentCssPath** key in the configuration properties file to the application specific CSS file location.

10.4. Implementation

The following classes implement the APP Service Document support:

- **com.hp.symphony.server.internal.resources.RootResource** – generates the XML (application/atomsvc+xml) representation of the APP Service Document.
- **com.hp.symphony.server.internal.resources.HtmlServiceDocumentResource -** generates the HTML (text/html) representation of the APP Service Document.

11. Spring Integration

Symphony 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 Symphony from the Spring context. When working with Spring, Symphony defines a core spring context that contains customization hooks, enabling easy customization that would otherwise require coding.

11.1. Spring Registration

Spring makes it convenient to register resources and providers as spring beans.

11.1.1. 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 Symphony 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/symphonyCoreContext-server.xml
               classpath:mycontext.xml
  </param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

11.1.2. Registering Resources and Providers

Symphony provides the com.hp.symphony.spring.Registrar class in order to register resources and providers through a Spring context. The Registrar class extends the SymphonyApplication 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 SymphonyApplication 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 SymphonyApplication

Refer to chapter 3, section 3.4 for more information on **Priorities**.

```
<bean class="com.hp.symphony.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>
```

11.2. Custom Properties File Definition

Symphony 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.PropertyPlaceholderConfig
urer">
  <property name="ignoreUnresolvablePlaceholders" value="true" />
  <property name="order" value="1" />
  <property name="propertiesArray">
    <list>
      <props>
       <prop
key="symphony.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 Symphony 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.

11.3. Customizing Media-Type Mappings

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

> Refer to chapter 3, section 3.7 for more information on **Media-Type Mapping**.

```
<bean id="custom.MediaTypeMapper"
class="com.hp.symphony.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.PropertyPlaceholderConfig
urer">
  <property name="ignoreUnresolvablePlaceholders" value="true" />
```

```
  <property name="order" value="1" />
  <property name="propertiesArray">
    <list>
      <props>
        <prop
key="symphony.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.

Note

customConfigurer

The order is set to "1". This makes the customConfigurer run before the default Symphony 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.

11.4. Customizing Alternative Shortcuts

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

> Refer to chapter 3, section 3.8 for more information on **Alternative Shortcuts Mappings**.

11.4.1. 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.PropertyPlaceholderConfigur
er">
  <property name="ignoreUnresolvablePlaceholders" value="true" />
  <property name="order" value="1" />
  <property name="propertiesArray">
    <list>
      <props>
        <prop key="symphony.alternateShortcutsMap">custom.Shortcuts</prop>
      </props>
    </list>
  </property>
</bean>
```

11.4.2. Spring Context File

Defines the map of the shortcuts in the Spring context.

12. WebDAV Extension

Symphony 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.

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

12.1. 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 **com.hp.symphony.webdav.model** package.

12.2. WebDAV Classes

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

12.2.1. 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.

12.2.2. 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.

12.3. 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.

12.4. 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.

12.4.1. 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.

12.4.2. 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 com.hp.symphony.common.model.synd.SyndFeed is created and passed to the WebDAVResponseBuilder.propfind() method in order to create the actual response.

12.4.3. Manual Creation

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

1. Create a new com.hp.symphony.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.
3. Return the Response instance from the resource method

13. Handler Chains

The Symphony 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 **com.hp.symphony.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.

13.1. Handlers

There are two types of handlers:

- System Handler
- User Handler

System Handlers are the handlers that implement the core engine of the Symphony runtime. The Symphony runtime will not function correctly if any of the system handlers are removed from the chain.

User Handlers 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 Symphony.

> Refer to chapter 2, section 2.3.1 for more details on **User Handlers Customization**.

13.2. Message Context

The **MessageContext** allows the following:

- Allows handlers to access and manipulate the current request information
- Allows handlers to maintain a state by setting attributes on the message context, as the handlers themselves are singletons and therefore stateless
- Allows handlers to pass information to other handlers on the chain

13.3. 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 **com.hp.symphony.server.handlers.RequestHandler** interface.

13.3.1. 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.

Request Handlers

Table 29: Request Handlers

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

13.3.2. User Request Handlers

User request handlers are inserted before the InvokeMethodHandler handler.

> Refer to chapter 2, section 2.3.1 for more details on **User Handlers Customization**.

13.4. 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 **com.hp.symphony.server.handlers.ResponseHandler** interface.

13.4.1. 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.

Response Handlers

Table 30: Response Handlers

Handler	Description
PopulateResponseStatus Handler	Determines the response status code, according to the JAX-RS spec
PopulateResponseMedia TypeHandler	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.

13.4.2. User Response Handlers

User response handlers are inserted before the FlushResultHandler handler. Symphony 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.

Refer to chapter 2, section 2.3.1 for more details on **User Handlers Customization**.

13.5. 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 **com.hp.symphony.server.handlers.ResponseHandler** interface.

13.5.1. 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

Table 31: 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

13.5.2. User Error Handlers

User error handlers are inserted before the FlushResultHandler handler.

Refer to chapter 2, section 2.3.1 for more details on **User Handlers Customization**.

13.6. Request Processing

The following details how the Symphony 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.
6. 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.

14. Symphony Client

The following chapter describes the Symphony Client and provides a detailed description of the Symphony Client component and its functionality.

14.1. This chapter contains the following sections

- Main Features
- High Level Architecture Overview
- Getting Started with the Symphony Client
- Configuration
- Handlers

14.2. Symphony Client Overview

The Symphony Client is an easy-to-use, high level Java API for writing clients that consume HTTP-based RESTful Web Services. It 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 Symphony Client also provides a Handlers mechanism that enables the manipulation of HTTP request/response messages.

14.3. Main Features

The Symphony Clients main features are as follows:

- Utilizes JAX-RS Providers for resource serialization and deserialization
- Provides Java object models, such as Atom, APP, OpenSearch and Json 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

14.4. High Level Architecture Overview

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

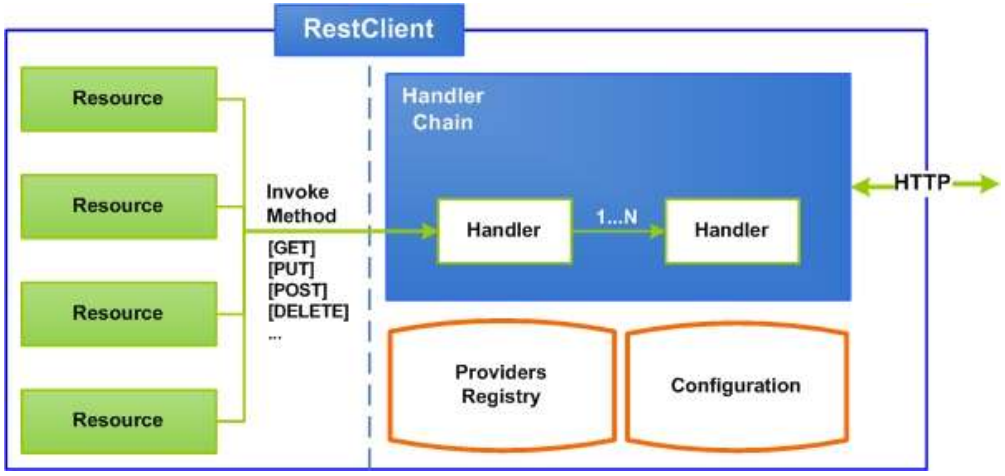

Figure 3: Symphony Client high-level view

The **RestClient** class is the Symphony 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.

14.5. Getting Started with the Symphony Client

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

14.5.1. 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. I order to start working with the **Symphony 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.

14.5.2. 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).

14.5.3. POST Atom 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");

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

14.5.4. Using ClientResponse

```
// 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

This example demonstrates how to use the ClientResponse object in order to deserialize the response entity. 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.

14.6. 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 & port
- Connect and read timeouts
- Redirect

14.6.1. Handler Configuration

```
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

This example demonstrates how to register a custom handler. 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.

14.6.2. Custom Provider Configuration

```
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

This example demonstrates how to register a custom entity provider. 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.

14.7. Client Handlers

The Symphony Client provides a Handlers mechanism that intercepts Http requests and responses. This mechanism is used to manipulate the request and response headers and allows the manipulation of the input and output entity streams by use of adapters.

An application develops custom handlers by implementing the **ClientHandler** interface. Custom handlers are registered via the ClientConfig configuration object. All registered handlers are invoked for every issued request, according to their registration order.

14.7.1. Custom Handlers

A custom handler implements the **ClientHandler** interface. The entry point of the handler is the **handle()** method. It receives the **ClientRequest** instance that encapsulates request information and allows for request data manipulation. It also receives the **HandlerContext** instance that is used to pass control to the next handler on the chain and to set input and output stream adapters.

Handlers are responsible for the flow control. It is handler's responsibility to call the next handler on the chain by invoking the doChain() method on the **HandlerContext** instance.

14.7.2. Custom Handler Implementation

A typical implementation of the "**handle**" method appears as follows:

```
public class MyHandler implements ClientHandler {

 public ClientResponse handle(ClientRequest rqs, HandlerContext ctx) {

   // Do something before request is issued to the web resource
1    rqs.getHeaders().add("CUSTOM-REQUEST-HEADER", "Foo-Request");

2    ClientResponse resp = ctx.doChain(request);

   // Do something before response is returned to the client
3    resp.getHeaders().add("CUSTOM-RESPONSE-HEADER", "Foo-Response");

4    return resp;

 }
}
```

Explanation

This example illustrates a typical implementation of the custom handler. First, the handler adds a custom header to the request as it appears in line 1, then it calls the next handler on the chain by invoking the doChain() method (line 2), adds a custom header to a response as it appears in line 3 and finally returns the response as it appears in line 4.

14.7.3. Input and Output Stream Adapters

The Symphony 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.

14.7.4. Stream Adapters Example

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

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);
  }}
```

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;
   }}
```

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