Red Hat JBoss Enterprise Application Platform 7.4

开发 Web 服务应用程序

为红帽 JBoss 企业应用平台开发 Web 服务应用程序的说明.

Last Updated: 2024-02-10

Red Hat JBoss Enterprise Application Platform 7.4 开发 Web 服务应用程序

为红帽 JBoss 企业应用平台开发 Web 服务应用程序的说明.

法律通告

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

摘要

本指南提供有关如何使用红帽 JBoss 企业应用平台开发 Web 服务应用程序的信息。

目录

提供有关 JBOSS EAP 文档的反馈 **4**

使开源包含更多 **5**

第 1 章 WEB 服务简介 **6**

第 2 章 开发 JAKARTA RESTFUL WEB SERVICES WEB 服务 **7**
2.1. JAKARTA RESTFUL WEB SERVICES 应用 7
2.2. JAKARTA RESTFUL WEB 服务客户端 8
2.3. JAKARTA RESTFUL WEB 服务请求处理 13
2.4. 查看 RESTEASY 端点 16
2.5. 使用 REGISTRYSTATSRESOURCE 查看 RESTEASY 端点 18
2.6. 基于 URL 协商 18
2.7. 内容划分和提供程序 19
2.8. 使用 JAKARTA JSON 处理 55
2.9. RESTEASY/JAKARTA ENTERPRISE BEANS 集成 58
2.10. SPRING 集成 60
2.11. JAKARTA 上下文和依赖注入集成 61
2.12. RESTEASY FILTERS 和 INTERCEPTORS 62
2.13. 日志记录 RESTEASY PROVIDERS 和 INTERCEPTORS 70
2.14. 异常处理 71
2.15. 保护 JAKARTA RESTFUL WEB 服务 WEB 服务 73
2.16. 异步作业服务 RESTEASY 76
2.17. RESTEASY JAVASCRIPT API 79
2.18. 用于修改资源元数据的 RESTEASY SPI 81
2.19. MICROPROFILE REST 客户端 82
2.20. 对 COMPLETIONSTAGE 类型的支持 88
2.21. 扩展 RESTEASY 支持异步请求处理和 REACTIVE RETURN 类型 89

第 3 章 开发 JAKARTA XML WEB 服务 **95**
3.1. 使用 JAKARTA XML WEB 服务工具 95
3.2. JAKARTA XML WEB 服务端点 101
3.3. JAKARTA XML WEB SERVICES WEB SERVICE 客户端 106
3.4. 配置 WEB 服务子系统 115
3.5. 分配客户端和端点配置 121
3.6. 为 WEB 服务应用设置模块依赖项 128
3.7. 配置 HTTP 超时 130
3.8. 保护 JAKARTA XML WEB 服务 131
3.9. JAKARTA XML WEB 服务日志记录 148
3.10. 启用 WEB 服务寻址(WS-ADDRESSING) 149
3.11. 启用 WEB 服务可靠的消息传递 150
3.12. 指定 WEB 服务策略 151
3.13. APACHE CXF 集成 152

附录 A. 参考资料 **160**
A.1. JAKARTA RESTFUL WEB SERVICES/RESTEASY ANNOTATIONS 160
A.2. RESTEASY 配置参数 162
A.3. RESTEASY JAVASCRIPT API 参数 165
A.4. RESTEAS.REQUEST 类成员 165
A.5. RESTEASY 异步作业服务配置参数 166
A.6. JAKARTA XML WEB 服务工具 167
A.7. JAKARTA XML WEB 服务通用 API 参考 171

A.8. 高级 WS-TRUST 场景 174

提供有关 JBOSS EAP 文档的反馈

要报告错误或改进文档，请登录到 Red Hat JIRA 帐户并提交问题。如果您没有 Red Hat Jira 帐户，则会提示您创建一个帐户。

流程

1. 单击以下链接 以创建 ticket。
2. 请包含 **文档 URL**、**章节编号** **并描述问题**。
3. 在 **Summary** 中输入问题的简短描述。
4. 在 **Description** 中提供问题或功能增强的详细描述。包括一个指向文档中问题的 URL。
5. 点 **Submit** 创建问题，并将问题路由到适当的文档团队。

使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 CTO Chris Wright 信息。

第 1 章 WEB 服务简介

Web 服务可提供一种标准方式，在不同软件应用之间进行互操作。每个应用程序都可以在各种平台和框架上运行。

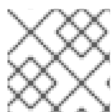
Web 服务促进了内部异构子系统通信。互操作性提高了服务重用性，因为不需要为各种环境重写函数。

第 2 章 开发 JAKARTA RESTFUL WEB SERVICES WEB 服务

Jakarta RESTful Web 服务是 RESTful Web 服务的 Jakarta EE API。它提供对使用表示状态转移或使用注释"REST"构建 Web 服务的支持。这些注释简化了将 Java 对象映射到 Web 资源的过程。

RESTEasy 是红帽 JBoss 企业应用平台 7 实施 Jakarta RESTful Web 服务。JBoss EAP 7.3 及更高版本符合 Jakarta 企业 Web Services 1.4 和 Jakarta RESTful Web Services 2.1 Jakarta EE 规范。它们还为规范提供了额外的功能。

要开始使用 Jakarta RESTful Web 服务，请参阅红帽 JBoss 企业应用平台 7 随附的 **helloworld-rs**、**jaxrs-client** 和 Kitchen **sink** 快速入门。



注意

JBoss EAP 不支持 **resteasy-crypto**、**resteasy-yaml-provider** 和 **jose-jwt** 模块。

2.1. JAKARTA RESTFUL WEB SERVICES 应用

在创建供应商和 Web 资源时，您可以使用以下命令来声明它们：

- 不带 **web.xml 文件的 javax.ws.rs.core.Application 的简单子类别.**
- 使用 **web.xml 文件.**
- 子类 **javax.ws.rs.core.Application** 并提供自定义实施.

2.1.1. Simple Subclassing **javax.ws.rs.core.Application**

您可以使用 **javax.ws.rs.core.Application** 类来创建声明这些提供程序和 Web 资源的子类。此类由 JBoss EAP 随附的 RESTEasy 库提供。

若要使用 **javax.ws.rs.core.Application** 配置资源或提供程序，只需创建一个扩展它的类并添加 **@ApplicationPath** 注释。

示例：应用程序类

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/root-path")
public class MyApplication extends Application
{
}
```

2.1.2. 使用 **web.xml**

或者，如果您不想创建扩展 **javax.ws.rs.core.Application** 的类，您可以将以下内容添加到您的 **web.xml** 文件中：

示例：web.xml

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
  </servlet>
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/root-path/*</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

2.1.3. 带有自定义实现的子类 javax.ws.rs.core.Application

在子类 **javax.ws.rs.core.Application** 时，您可以选择为任何现有方法提供自定义实施。**getClasses** 和 **getSingletons** 方法返回一系列必须包含在发布的 Jakarta RESTful Web Services 应用程序中的类或单例集合。

- 如果 **getClasses** 和 **getSingletons 返回了** 非空集合，则 Jakarta RESTful Web Services 应用中 *只会* 发布这些类和单例。
- 如果 **getClasses** 和 **getSingletons 都** 返回一个空集合，则 Web 应用中打包 *的所有* 根资源类和提供程序 *都包含* 在 Jakarta RESTful Web Services 应用中。RESTEasy 随后将自动发现这些资源。

2.2. JAKARTA RESTFUL WEB 服务客户端

2.2.1. Jakarta RESTful Web Services 客户端 API

Jakarta RESTful Web Services 2.0 引入了一个新的客户端 API，用于将 HTTP 请求发送到远程 RESTful Web 服务。这是一个 *流畅* 的请求构建 API，包含 3 个主要类：

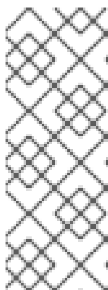
- **客户端**
- **Web 目标**
- **响应**

客户端 接口是 Web 目标实例的构建器。**WebTarget** 代表用于构建子资源 Web 目标或调用请求的不同 URL 或 URL 模板。

可以通过两种方式创建客户端：标准方式，或使用 **ResteasyClientBuilder** 类。使用 **ResteasyClientBuilder** 类的优点是它提供了一些额外的帮助程序方法来配置您的客户端。

ResteasyClientBuilder 类提供这些帮助程序方法，但该类特定于 JBoss EAP API。如果要将应用程序迁移到新服务器，您必须重新构建应用程序。**ResteasyClientBuilder** 类依赖于 RESTEasy，并且类不可移植。

创建客户端的标准方式同时遵循 Jakarta RESTful Web 服务和 Jakarta EE API 规范，可在 Jakarta RESTful Web 服务实施中移植。



注意

为确保 Jakarta RESTful Web Services 应用保持可移植性，请遵循 Jakarta EE API 规范，并在可能的情况下使用 Jakarta EE API 应用。如果用例不支持使用 Jakarta EE API，则仅使用特定于 JBoss 的 API。

遵循这些准则可以帮助减少将应用迁移到其他服务器或新的 Jakarta EE 兼容 JBoss 实施时可能出现的问题数量。

使用标准方法创建客户端

以下示例显示了创建客户端的标准方法之一：

```
Client client = ClientBuilder.newClient();
```

另外，您可以使用另一种标准方法来创建客户端，如下例所示：

```
Client client = ClientBuilder.newBuilder().build();
WebTarget target = client.target("http://foo.com/resource");
Response response = target.request().get();
String value = response.readEntity(String.class);
response.close();  // You should close connections!
```

使用 ResteasyClientBuilder 类创建客户端

以下示例演示了使用 **ResteasyClientBuilder** 类来创建客户端：

```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target("http://foo.com/resource");
```

通过 Jakarta RESTful Web Services 2.1， 您可以向 **ClientBuilder** 类添加两种超时方法。超时方法是符合规格的方法， 您可以使用它们， 而不使用 RESTEasy 方法。

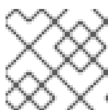
以下 **ClientBuilder** 规格兼容方法替换了一些已弃用的 RESTEasy 方法：

- **connectTimeout** 方法替换了 create **ConnectionTimeout** 方法。
  connectTimeout 方法决定了客户端在进行新服务器连接时必须等待的时长。
- **readTimeout** 方法替换了 **socketTimeout** 方法。
  readTimeout 方法决定了客户端必须等待来自服务器的响应的时长。

以下示例显示了 **connectTimeout 和 readTimeout 方法** 的指定值：

```
import javx.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;

Client client = ClientBuilder.newBuilder()
        .connectTimeout(100, TimeUnit.SECONDS)
        .readTimeout(2, TimeUnit.SECONDS)
        .build();
```

请注意， **readTimeout** 适用于现有连接上已执行的请求。



注意

将 timeout 参数的值设置为零会导致服务器无限期等待。

RESTEasy 自动加载一组默认提供程序，其中包含 **META-INF/services/javax.ws.rs.ext.Providers** 文件中列出的所有类。另外，您可以通过方法调用 **Client.configuration（）** 提供的配置对象手动注册其他提供程序、过滤器和拦截器。通过配置，您可以设置可能需要的配置属性。

每个 **Web 目标** 都有一个配置实例，它继承了与父实例注册的组件和属性。这可让您为每个目标资源设置特定的配置选项，例如用户名和密码。

其它资源

- 如需有关 **ResteasyClientBuilder** 类及其方法的更多信息，请参见 Class ResteasyClientBuilder。

使用 RESTEasy 客户端类
您必须将 RESTEasy 客户端的以下依赖项添加到 Maven **pom.xml** 文件中：

```
<dependency>
 <groupId>org.jboss.resteasy</groupId>
 <artifactId>resteasy-client</artifactId>
 <version>VERSION_IN_EAP</version>
 </dependency>
```

有关使用 RESTEasy 客户端类 **的工作示例，请参见 JBoss EAP 附带的 jaxrs-client 和 resteasy-jaxrs-client** 快速入门。

客户端过滤器
客户端具有两种过滤器：

ClientRequestFilter

ClientRequestFilter 在通过线路将 HTTP 请求发送到服务器之前运行。 **ClientRequestFilter** 也被允许中止请求执行并提供修剪的响应，而无需通过线路传输到服务器。

ClientResponseFilter

ClientResponseFilter 在从服务器收到响应后运行，但在取消托管响应正文之前。**ClientResponseFilter** 可以在将响应对象提交到应用代码之前对其进行修改。以下示例演示了这些概念：

```
// execute request filters
for (ClientRequestFilter filter : requestFilters) {
 filter.filter(requestContext);
 if (isAborted(requestContext)) {
   return requestContext.getAbortedResponseObject();
  }
}

// send request over the wire
response = sendRequest(request);

// execute response filters
for (ClientResponseFilter filter : responseFilters) {
 filter.filter(requestContext, responseContext);
}
```

将客户端过滤器注册到客户端请求
以下示例演示了如何将客户端过滤器注册到客户端请求：

```
client = ClientBuilder.newClient();
WebTarget base = client.target(generateURL("/") + "get");
base.register(ClientExceptionsCustomClientResponseFilter.class).request("text/plain").get();
```

客户端缓存
RESTEasy 能够设置客户端缓存。此缓存查找随服务器响应发回的 cache-control 标头。如果 cache-control 标头指定允许客户端缓存响应，RESTEasy 会将它缓存在本地内存中。

```
ResteasyWebTarget target = client.target(generateBaseUrl());
target.register(BrowserCacheFeature.class);
```

块编码支持
RESTEasy 为客户端 API 提供指定请求应在 *区块传输* 模式中发送的功能。可以通过两种方式指定 *区块传输* 模式，如下所示：

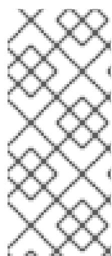
- 您可以将 **org.jboss.resteasy.client.jaxrs.ResteasyWebTarget** 配置为以区块模式发送所有请求：

```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target("http://localhost:8081/test");
target.setChunked(b.booleanValue());
Invocation.Builder request = target.request();
```

- 另外，您还可以将特定请求配置为以块模式发送：

```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target("http://localhost:8081/test");
ClientInvocationBuilder request = (ClientInvocationBuilder) target.request();
request.setChunked(b);
```

与 **javax.ws.rs.client.Invocation.Builder** 类不同，**org.jboss.resteasy.client.jaxrs.internal.ClientInvocationBuilder** 是 RESTEasy 类。



注意

以块模式发送请求的能力取决于底层传输层。特别是，它取决于所使用的 **org.jboss.resteasy.client.jaxrs.ClientHttpEngine** 类的实施。目前，只有 **ApacheHttpClient43Engine** 和之前实施 **ApacheHttpClient4Engine** 的默认实施支持区块模式。这两个都位于 **org.jboss.resteasy.client.jaxrs.engines 软件包中**。如需更多信息，请参阅 通过 HTTP 客户端实施 RESTEasy 部分。

2.2.2. 使用 HTTP 客户端实施 RESTEasy

默认情况下，在 RESTEasy 中处理客户端和服务器之间的网络通信。它使用 Apache **Http Components** 项目中的 **Http Client**。RESTEasy 客户端框架和网络之间的接口由 **ClientHttpEngine** 接口定义。

RESTEasy 随附此接口的四种实施：默认实施是 **ApacheHttpClient43Engine**。此实施使用 Apache 4.3。

ApacheHttpClient4Engine 是一种使用比 Apache 4.3 更早的版本的实现。此类提供向后兼容性。RESTEasy 根据 Apache 版本检测自动选择这两个 **ClientHttpEngine** 实施之一。**InMemoryClientEngine** 是一种将请求分配到同一 JVM 中的服务器的实施，而 **URLConnectionEngine** 则是使用 **java.net.HttpURLConnection** 的一种实施。

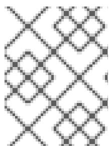
客户端执行器可以传递给特定的 **ClientRequest** ：

```
ResteasyClient client = new
ResteasyClientBuilder().httpEngine(engine).build();
```

RESTEasy 和 **HttpClient** 做出默认决策以使用客户端框架，而无需引用 **HttpClient**。但是，在某些应用中，可能需要深入查看 **HttpClient** 详细信息。**ApacheHttpClient43Engine** 和 **ApacheHttpClient4Engine** 可以提供 **org.apache.http.client.HttpClient** 和 **org.apache.http.protocol.HttpContext** 的实例，它们将额外的配置详细信息传输到 **HttpClient** 层。例如，身份验证可以配置如下：

```
// Configure HttpClient to authenticate preemptively
// by prepopulating the authentication data cache.

// 1. Create AuthCache instance
AuthCache authCache = new BasicAuthCache();

// 2. Generate BASIC scheme object and add it to the local auth cache
AuthScheme basicAuth = new BasicScheme();
authCache.put(new HttpHost("sippycups.bluemonkeydiamond.com"), basicAuth);

// 3. Add AuthCache to the execution context
BasicHttpContext localContext = new BasicHttpContext();
localContext.setAttribute(ClientContext.AUTH_CACHE, authCache);

// 4. Create client executor and proxy
HttpClient httpClient = HttpClientBuilder.create().build();
ApacheHttpClient4Engine engine = new ApacheHttpClient4Engine(httpClient, localContext);
ResteasyClient client = new ResteasyClientBuilder().httpEngine(engine).build();
```

HttpContextProvider 是 RESTEasy 提供的接口，您可以使用它为 Apache **Http Client43Engine 和 ApacheHttpClient4Engine 实施提供自定义 Http Context**。



注意

了解 *释放* 连接 *和关闭* 连接之间的区别非常重要。释放连接使其可以重复利用。关闭连接会释放其资源，使其无法使用。

RESTEasy 在没有通知的情况下释放连接。唯一的计数器示例是响应是 **InputStream** 实例，它必须显式关闭。

另一方面，如果调用的结果是 Response 实例，则必须使用 **Response .close（）** 方法来释放连接。

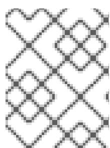
```
WebTarget target = client.target("http://localhost:8081/customer/123");
Response response = target.request().get();
System.out.println(response.getStatus());
response.close();
```

您可以在一个 **试用的 块中执行此操作**。释放连接使其可用于另一用途。它通常不关闭套接字。

如果创建了任何开放的套接字，**ApacheHttpClient4Engine.finalize（）** 将关闭它一直使用的 **HttpClient**。依靠 JDK 调用 **finalize（）** 并不安全。如果 **HttpClient** 传递给 **ApacheHttpClient4Executor**，用户必须关闭连接，如下所示：

```
HttpClient httpClient = new HttpClientBuilder.create().build();
ApacheHttpClient4Engine executor = new ApacheHttpClient4Engine(httpClient);
```

```
...
httpClient.getConnectionManager().shutdown();
```



注意

如果 **ApacheHttpClient4Engine** 创建了自己的 **HttpClient** 实例，则无需等待 **finalize（）** 关闭打开套接字。**ClientHttpEngine** 接口具有一个 **close（）** 方法来实现这一目的。

最后，如果 **javax.ws.rs.client.Client** 类已自动创建引擎，请调用 **Client.close(**)。这个调用会清理任何套接字连接。

2.2.2.1. HTTP 重定向

基于 Apache H **ttpClient 的 ClientHttpEngine** 实施支持 HTTP 重定向。默认情况下禁用此资源。您可以通过将 **setFollowRedirects** 方法设置为 **true 来启用** 此功能，如下所示：

```
ApacheHttpClient43Engine engine = new ApacheHttpClient43Engine();
engine.setFollowRedirects(true);
Client client = new ResteasyClientBuilder().httpEngine(engine).build();
```

2.3. JAKARTA RESTFUL WEB 服务请求处理

2.3.1. 异步 HTTP 请求处理

异步请求处理允许您使用非阻塞输入和输出来处理单个 HTTP 请求，如果需要，也可以在单独的线程中处理。

考虑 AJAX 聊天客户端，您要在其中从客户端和服务器推送和拉取。此场景使客户端在服务器的套接字上长时间阻止，等待新的消息。如果同步 HTTP 处理（其中服务器在传入和传出输入和输出上阻塞），则每个客户端连接使用一个单独的线程。这种请求处理模式消耗了大量内存和宝贵的线程资源。

异步处理分隔连接接受和请求处理操作。它分配两个不同的线程：一个用于接受客户端连接；另一个用于处理大量耗时的操作。在这个模型中，容器的工作方式如下：

1. 它分配线程接受客户端连接，这是接收器。
2. 然后，它将请求交给处理线程，即工作程序。
3. 最后，它会释放接收器线程。

结果由 worker 线程发送回客户端。因此，客户端的连接保持开放，从而提高服务器的吞吐量和可扩展性。

2.3.1.1. 异步 NIO 请求处理

RESTEasy 的默认异步引擎实施类是 **ApacheHttpAsyncClient4Engine**。它在 Apache **Http Components 的 Http AsyncClient** 上构建，后者使用非阻塞 IO 模型内部分配请求。

您可以通过调用 **ResteasyClientBuilder** 类中的 **useAsyncHttpEngine** 方法将异步引擎设置为活跃引擎：

```
Client asyncClient = new ResteasyClientBuilder().useAsyncHttpEngine()
                .build();
```

```
Future<Response> future = asyncClient
                  .target("http://localhost:8080/test").request()
                  .async().get();
Response res = future.get();
Assert.assertEquals(HttpResponseCodes.SC_OK, res.getStatus());
String entity = res.readEntity(String.class);
```

2.3.1.2. 服务器异步响应处理

在服务器端，异步处理涉及暂停原始请求线程，并在不同的线程中启动请求处理，后者释放原始服务器端线程，以接受其他传入的请求。

2.3.1.2.1. AsyncResponse API

Jakarta RESTful Web Services 2.1 规范使用两个类定义异步 HTTP 支持：**@Suspended** 注释和 **AsyncResponse** 接口。

将 **AsyncResponse** 作为参数注入到 Jakarta RESTful Web Services 方法，提示 RESTEasy 将 HTTP 请求和响应与当前执行的线程分离。这样可确保当前线程不会尝试自动处理响应。

AsyncResponse 是回调对象。调用其中一个 restore **（）方法** 的操作会导致响应发回到客户端，并且终止 HTTP 请求。以下是异步处理的示例：

```
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.AsyncResponse;

@Path("/")
public class SimpleResource {
  @GET
  @Path("basic")
  @Produces("text/plain")
  public void getBasic(@Suspended final AsyncResponse response) throws Exception {
    Thread t = new Thread() {
      @Override
      public void run() {
        try {
          Response jaxrs = Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
          response.resume(jaxrs);
        }
        catch (Exception e) {
          e.printStackTrace();
        }
      }
    };
    t.start();
  }
}
```

2.3.1.3. AsyncInvoker Client API

同样，在客户端上，异步处理可防止阻止请求线程，因为不需要花时间等待来自服务器的响应。例如，发出请求的线程也可以更新用户界面组件。如果线程被阻止等待响应，用户感知的应用性能将会受到影响。

2.3.1.3.1. 使用将来

在下面的代码片段中，**get（）** 方法在 **async（）** 方法上调用，而不是请求。这会将调用机制从同步更改为异步。**async（）** 方法不同步响应，而是返回一个 **将来** 的对象。当您 **调用 get（）** 方法时，调用会被阻止，直到响应就绪。当响应就绪时，会返回 if **.get（）** 方法。

```
import java.util.concurrent.Future;
import javax.ws.rs.client.Client;
...

@Test
public void AsyncGetTest() throws Exception {
    Client client = ClientBuilder.newClient();
    Future<String> future = client.target(generateURL("/test")).request().async().get(String.class);
    String entity = future.get();
    Assert.assertEquals("get", entity);
}
```

2.3.1.3.2. 使用 InvocationCallback

通过 **AsyncInvoker** 接口，您可以在异步调用准备好处理时注册回调用的对象。**InvocationCallback** 接口提供了两种方法：**completed（）** 和 **failed（）**。当处理成功完成并且收到响应时，会调用 **completed（）** 方法。相反，每当请求处理不成功时，会调用 **failed（）** 方法。

```
import javax.ws.rs.client.InvocationCallback;
...

@Test
public void AsyncCallbackGetTest() throws Exception {
    Client client = ClientBuilder.newClient();
 final CountDownLatch latch = new CountDownLatch(1);
        Future<Response> future = client.target(generateURL("/test")).request().async().get(new
InvocationCallback<Response>() {
         @Override
             public void completed(Response response) {
                 String entity = response.readEntity(String.class);
                 Assert.assertEquals("get", entity);
                 latch.countDown();
             }

             @Override
             public void failed(Throwable error) {
             }
         });
         Response res = future.get();
         Assert.assertEquals(HttpResponseCodes.SC_OK, res.getStatus());
         Assert.assertTrue("Asynchronous invocation didn't use custom implemented Invocation
callback", latch.await(5, imeUnit.SECONDS));
}
```

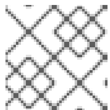
2.3.2. 自定义 RESTEasy Annotations

由于字节码中添加了参数名称，您不再需要在以下注释中指定参数名称：@ **PathParam**、**@QueryParam**、**@Form Param**、**@CookieParam**、**@HeaderParam**、@**HeaderParam** 和 **@MatrixParam**。为此，您必须在具有可选 value 参数的不同软件包中切换到名称相同的新注解。您可以按照以下步骤实现：

1. 导入 **org.jboss.resteasy.annotations.jaxrs** 软件包，以替换 Jakarta RESTful Web 服务规格中的注释。
2. 配置构建系统，以在字节码中记录方法参数名称。
   Maven 用户可以通过将 **maven.compiler.parameters** 设置为 **true** 来在字节码中启用记录方法参数名称：

```
<properties>
   <maven.compiler.parameters>true</maven.compiler.parameters>
</properties>
```

3. 如果名称与注释的变量的名称匹配，则删除注解值。



注意

您可以省略注释的方法参数的注解名称，以及注释的字段或 JavaBean 属性。

例如，请考虑以下用法：

```
import org.jboss.resteasy.annotations.jaxrs.*;

@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam String isbn) {
      // search my database and get a string representation and return it
   }
}
```

如果您标注的变量的名称与 path 参数不同，您可以指定名称，如下所示：

```
import org.jboss.resteasy.annotations.jaxrs.*;

@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") String id) {
      // search my database and get a string representation and return it
   }
}
```

2.4. 查看 RESTEASY 端点

您可以使用 **jaxrs** 子系统 **的 read-resource** 操作来查看每个 RESTEasy 端点的结构化输出。下面提供了管理 CLI 命令和预期结果的示例：

```
/deployment=DEPLOYMENT_NAME/subsystem=jaxrs/rest-
resource=org.jboss.as.quickstarts.rshelloworld.HelloWorld:read-resource(include-runtime=true)
{
```

```
    "outcome" => "success",
    "result" => {
        "resource-class" => "org.jboss.as.quickstarts.rshelloworld.HelloWorld",
        "rest-resource-paths" => [
            {
                "resource-path" => "/hello/json",
                "consumes" => undefined,
                "produces" => [
                    "application/json",
                    "text/plain"
                ],
                "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.HelloWorld.getHelloWorldJSON()",
                "resource-methods" => [
                    "POST /wildfly-helloworld-rs/rest/hello/json",
                    "GET /wildfly-helloworld-rs/rest/hello/json"
                ]
            },
            {
                "resource-path" => "/hello/xml",
                "consumes" => undefined,
                "produces" => ["application/xml"],
                "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.HelloWorld.getHelloWorldXML(@QueryParam java.lang.String
name = 'LGAO')",
                "resource-methods" => ["GET /wildfly-helloworld-rs/rest/hello/xml"]
            }
        ],
        "sub-resource-locators" => [{
            "resource-class" => "org.jboss.as.quickstarts.rshelloworld.SubHelloWorld",
            "rest-resource-paths" => [
                {
                    "resource-path" => "/hello/subMessage/",
                    "consumes" => undefined,
                    "produces" => undefined,
                    "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.SubHelloWorld.helloInSub()",
                    "resource-methods" => ["GET /wildfly-helloworld-rs/rest/hello/subMessage/"]
                },
                {
                    "resource-path" => "/hello/subMessage/subPath",
                    "consumes" => undefined,
                    "produces" => undefined,
                    "java-method" => "java.lang.String
org.jboss.as.quickstarts.rshelloworld.SubHelloWorld.subPath()",
                    "resource-methods" => ["GET /wildfly-helloworld-rs/rest/hello/subMessage/subPath"]
                }
            ],
            "sub-resource-locators" => undefined
        }]
    }
}
```

在上例中，输出信息按 resource **-class** 分组，并根据 **resource-path 进行排序** ：

- **resource-path** 是访问端点的地址。

- **resource-class** 定义端点定义的类。
- **rest-resource-paths** 包括定义资源路径、HTTP 方法、消耗和生成端点的 Java 方法。
- **java-method** 指定 Java 方法的名称及其参数。它还包含下列 Jakarta RESTful Web Services 注释（若已定义）：**@PathParam**、**@HeaderParam**、**@MatrixParam**、**@ CookieParam**、@**FormParam** 和 **@DefaultValue**。

另外，您可以使用未定义 **rest -resource 参数的 read -resource** 操作，并获取有关所有端点的信息，如下例所示：

```
/deployment=DEPLOYMENT_NAME/subsystem=jaxrs:read-resource(include-runtime=true,recursive=true)
```

2.5. 使用 REGISTRYSTATSRESOURCE 查看 RESTEASY 端点

您可以从 **RegistryStatsResource** 资源获取应用的 RESTEasy 端点的信息。

流程

1. 通过在应用程序的部署描述符 **web.xml** 文件中添加以下 XML 片断来注册 **RegistryStatsResource** ：

```
<context-param>
    <param-name>resteasy.resources</param-name>
    <param-value>org.jboss.resteasy.plugins.stats.RegistryStatsResource</param-value>
</context-param>
```

2. 查看应用的 RESTEasy 端点：
   - 使用 CLI：
     - 获取 XML 的结果：

```
$ curl http://localhost:8080/{APPLICATION_PREFIX_URL}/resteasy/registry
```

     - 获取 JSON 的结果：

```
$ curl http://localhost:8080/{APPLICATION_PREFIX_URL}/resteasy/registry -H
"Accept: application/json"
```

   - 使用 Web 浏览器：

```
http://localhost:8080/{APPLICATION_PREFIX_URL}/resteasy/registry
```

2.6. 基于 URL 协商

2.6.1. 将扩展映射到介质类型

些客户端（如浏览器）无法使用 **Accept 和 Accept -Language** 标头来协商表示介质类型或语言。RESTEasy 可以将文件名后缀映射到介质类型和语言，以处理此问题。

要使用 **web.xml** 文件将介质类型映射到文件扩展，您需要添加 **resteasy.media.type.mappings** 上下文 param 和映射列表作为 **param-value**。该列表用逗号分开，并使用冒号(:)来分隔文件扩展名和媒体类型。

web.xml 映射文件扩展示例到介质类型

```
<context-param>
    <param-name>resteasy.media.type.mappings</param-name>
    <param-value>html : text/html, json : application/json, xml : application/xml</param-value>
</context-param>
```

在本例中，映射 **http://localhost:8080/my-application/test** 的以下 URL 变体：

- **http://localhost:8080/my-application/test.html**
- **http://localhost:8080/my-application/test.json**
- **http://localhost:8080/my-application/test.xml**

2.6.2. 将扩展映射到语言

些客户端（如浏览器）无法使用 **Accept 和 Accept -Language** 标头来协商表示介质类型或语言。RESTEasy 可以将文件名后缀映射到介质类型和语言，以处理此问题。按照以下步骤，在 **web.xml** 文件中将语言映射到文件扩展名。

要使用 **web.xml** 文件将介质类型映射到文件扩展，您需要添加 **resteasy.language.mappings** 上下文 param 和映射列表作为 **param-value**。该列表用逗号分开，并使用冒号(:)来分隔文件扩展名和语言类型。

web.xml 映射文件扩展示例到语言类型

```
<context-param>
    <param-name>resteasy.language.mappings</param-name>
    <param-value> en : en-US, es : es, fr : fr</param-name>
</context-param>
```

在本例中，映射 **http://localhost:8080/my-application/test** 的以下 URL 变体：

- **http://localhost:8080/my-application/test.en**
- **http://localhost:8080/my-application/test.es**
- **http://localhost:8080/my-application/test.fr**

2.7. 内容划分和提供程序

2.7.1. 默认供应商和默认的 Jakarta RESTful Web 服务内容转换

RESTEasy 可以自动托管和解封几个不同的消息正文。

表 2.1. 支持的介质类型和 Java 类型

介质类型	Java 类型
application/* +xml,text/* +xml,application/* +json,application/* +fastinfoset,application/ atom+*	jakarta XML Binding 注解的类
application/* +xml,text/* +xml	org.w3c.dom.Document
/	java.lang.String
/	java.io.InputStream
text/plain	用于输出的原语、java.lang.String 或具有 String 构造器或静态 valueOf(String)方法的任何类型
/	javax.activation.DataSource
/	java.io.File
/	byte
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

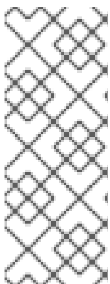
2.7.1.1. 文本介质类型和字符集

根据 Jakarta RESTful Web 服务规范，在编写响应时，实施必须遵循应用提供的字符集元数据。如果没有由应用程序指定字符集，或者应用程序指定了不支持的字符集，则实施必须使用 UTF-8 字符集。

相反，根据 HTTP 规范，发件人未提供显式 charset 参数时，将 **文本** 类型的媒体子类型定义为具有默认的 charset 值 **ISO-8859-1**。**ISO-8859-1** 或其子集以外的字符集中的数据必须使用适当的 charset 值进行标记。

如果没有通过资源或资源方法指定的字符集，RESTEasy 将 UTF-8 用作文本媒体类型的字符集。为此，RESTEasy 会将一个明确的 charset 参数添加到内容类型响应标头中。

要指定原始行为，其中 UTF-8 用于文本介质类型，但未附加显式 charset 参数，请将上下文参数 **resteasy.add.charset** 设置为 **false**。此参数的默认值为 **true**。



注意

文本介质类型包括：

- 类型为 **文本** 和任何子类型的介质类型.
- 类型为 **application** 和子类型（以 **xml** 开头的介质类型）这包括 **application/xml-external-parsed-entity** 和 **application/xml-dtd**。

2.7.2. 使用 @Provider 类构建内容

Jakarta RESTful Web 服务规范允许您插入自己的请求/响应正文阅读器和作者。为此，您可以为类标上 **@Provider**，再为读取器指定 **@Produces** 类型和 **@Consumes** 类型。您还必须实施 **MessageBodyReader/Writer** 接口。

使用 **@Provider** 标注的客户端提供程序必须为 Jakarta RESTful Web Services 容器运行时的每个客户端实例注册，以处理注释。为了避免意外或重复的客户端供应商注册出现问题，系统属性 **resteasy.client.providers.annotations.disabled** 会禁用带有 **@Provider** 标注的客户端供应商的默认处理。

RESTEasy **ServletContextLoader** 会自动扫描标有 **@Provider** 的类的 **WEB-INF/lib** 和 class 目录，或者您可以在 **web.xml** 文件中手动配置它们。

2.7.3. 提供程序实用程序类

javax.ws.rs.ext.Providers 是一个简单的可注入界面，允许您查找 **MessageBodyReaders**、Writers、**ContextResolvers** 和 **ExceptionMappers**。它对于实施嵌入其他随机内容类型的多部分提供程序和内容类型非常有用。

```
public interface Providers {
  <T> MessageBodyReader<T> getMessageBodyReader(Class<T> type, Type genericType,
Annotation annotations[], MediaType mediaType);
  <T> MessageBodyWriter<T> getMessageBodyWriter(Class<T> type, Type genericType, Annotation
annotations[], MediaType mediaType);
  <T extends="" throwable=""> ExceptionMapper<T> getExceptionMapper(Class<T> type);
  <T> ContextResolver<T> getContextResolver(Class<T> contextType, MediaType mediaType);
}
```

Providers 实例可以注入到 **MessageBodyReader 或 Writers** 中：

```
@Provider
@Consumes("multipart/fixed")
public class MultipartProvider implements MessageBodyReader {

  private @Context Providers providers;
  ...
}
```

2.7.4. 配置文档强制

XML 文档解析器受到 XXE(XML eXternal Entity)攻击，其中扩展外部实体会导致加载不安全的文件。例如，以下文档可能会导致加载 **/etc/passwd** 文件：

```
<!--?xml version="1.0"?-->
<!DOCTYPE foo
[<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<search>
 <user>bill</user>
 <file>&xxe;<file>
</search>
```

默认情况下，**org.w3c.dom.Document** 文档的 RESTEasy 内置 unmarshaller 不扩展外部实体。它将它们替换为空字符串。您可以将其配置为将外部实体替换为 DTD 中定义的值。这可以通过在 **web.xml 文件中将 resteasy.document.expand.entity. references** 上下文参数设置为 **true** 来实现。

示例：设置 resteasy.document.expand.entity.references 上下文参数

```
<context-param>
 <param-name>resteasy.document.expand.entity.references</param-name>
 <param-value>true</param-value>
</context-param>
```

另一种解决问题的方法是禁止 DTD，RESTEasy 默认这样做。可以通过将 **resteasy.document.secure.disableDTDs** 上下文参数设置为 **false** 来更改此行为。

示例：设置 resteasy.document.secure.disableDTDs Context Parameter

```
<context-param>
 <param-name>resteasy.document.secure.disableDTDs</param-name>
 <param-value>false</param-value>
</context-param>
```

当缓冲区被大型实体超额或太多属性时，文档也会受到 *服务攻击的 Denial 服务攻击* 的影响。例如，如果 DTD 定义了以下实体，则 **&foo6 的扩展会导致** 1,000,000 foos。

```
<!--ENTITY foo 'foo'-->
<!--ENTITY foo1 '&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;&foo;'-->
<!--ENTITY foo2 '&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;'-->
<!--ENTITY foo3 '&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;&foo2;'-->
<!--ENTITY foo4 '&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;&foo3;'-->
<!--ENTITY foo5 '&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;&foo4;'-->
<!--ENTITY foo6 '&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;&foo5;'-->
```

默认情况下，RESTEasy 限制各个实体的扩展数和属性数量。确切的行为取决于底层解析器。可以通过将 **resteasy.document.secure.process.feature** 上下文参数设置为 **false** 来关闭限制。

示例：设置 resteasy.document.secure.process.feature Context Parameter

```
<context-param>
 <param-name>resteasy.document.secure.processing.feature</param-name>
 <param-value>false</param-value>
</context-param>
```

2.7.5. 使用 MapProvider

您可以使用 **MapProvider** 接受并返回 Jakarta RESTful Web Services 资源映射。

示例：资源接受和返回映射

```
@Path("manipulateMap")
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces("application/x-www-form-urlencoded")
public MultivaluedMap<String, String> manipulateMap(MultivaluedMap<String, String> map) {
  //do something
  return map;
}
```

您还可以使用 客户端发送和接收 map 到 Jakarta RESTful Web 服务资源。

示例： Client

```
MultivaluedMap<String, String> map = new MultivaluedHashMap<String, String>();

//add values to the map...

Response response = client.target(generateURL("/manipulateMap"))
                    .request(MediaType.APPLICATION_FORM_URLENCODED_TYPE)
                    .post(Entity.entity(map, MediaType.APPLICATION_FORM_URLENCODED_TYPE));

String data = response.readEntity(String.class);

//handle data...
```

2.7.6. 将基于字符串的注解转换为对象

Jakarta RESTful Web Services **@*Param** 注释，包括 **@QueryParam**、**@MatrixParam**、**@HeaderParam**、**@PathParam** 和 **@FormParam**，在原始 HTTP 请求中以字符串形式表示。如果这些对象具有 **valueOf(String)** 静态方法或采用一个 **String** 参数的构造器，这些注入的参数可转换为对象。

如果您的类为 **valueOf（）** 方法或字符串构造器不存在或不适合 HTTP 请求，Jakarta RESTful Web 服务将提供 **javax.ws.rs.ext.ParamConverterProvider** 和 **javax.ws.rs.ext.ParamConverter**，以帮助将消息参数值转换为对应的自定义 Java 类型。**ParamConverterProvider** 必须通过编程方式在 Jakarta RESTful Web Services 运行时注册，必须标上 **@Provider** 注释，以便在提供商扫描阶段由 Jakarta RESTful Web Services 运行时自动发现。

例如：以下步骤演示了如何创建自定义 POJO 对象。从消息参数值（如 **@QueryParam**、@**PathParam**、**@MatrixParam**、**@HeaderParam** 转换为 POJO 对象的转换是通过实施 **ParamConverter Provider** 接口实现的。

1. 创建自定义 POJO 类。

```
public class POJO {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

2. 创建自定义 POJO Converter 类。

```
public class POJOConverter implements ParamConverter<POJO> {
  public POJO fromString(String str) {
    System.out.println("FROM STRNG: " + str);
    POJO pojo = new POJO();
    pojo.setName(str);
    return pojo;
```

```
    }

    public String toString(POJO value) {
      return value.getName();
    }
  }
```

3. 创建自定义 POJO Converter Provider 类。

```
public class POJOConverterProvider implements ParamConverterProvider {
  @Override
  public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType,
Annotation[] annotations) {
    if (!POJO.class.equals(rawType)) return null;
    return (ParamConverter<T>)new POJOConverter();
  }
}
```

4. 创建自定义 MyResource 类。

```
@Path("/")
public class MyResource {
  @Path("{pojo}")
  @PUT
  public void put(@QueryParam("pojo") POJO q, @PathParam("pojo") POJO pp,
@MatrixParam("pojo") POJO mp,
    @HeaderParam("pojo") POJO hp) {
    ...
  }
}
```

扩展 ParamConverter 的功能
在 Jakarta RESTful Web 服务语义中，**ParamConverter** 转换代表单个对象的单个字符串。RESTEasy 扩展语义，以允许 **ParamConverter** 解析多个对象的字符串表示，并生成 **List<T>**、**Set<T>**、Sorted **Set<T>**、 **数组** 或任何其他多值数据结构。

例如，请考虑以下资源：

```
@Path("queryParam")
public static class TestResource {
   @GET
   @Path("")
   public Response conversion(@QueryParam("q") List<String> list) {
      return Response.ok(stringify(list)).build();
   }
}

private static <T> String stringify(List<T> list) {
   StringBuffer sb = new StringBuffer();
   for (T s : list) {
      sb.append(s).append(',');
   }
   return sb.toString();
}
```

按照如下所示调用 **TestResource**，使用标准表示法：

```
@Test
public void testQueryParamStandard() throws Exception {
  ResteasyClient client = new ResteasyClientBuilder().build();
  Invocation.Builder request = client.target("http://localhost:8081/queryParam?
q=20161217&q=20161218&q=20161219").request();
  Response response = request.get();
  System.out.println("response: " + response.readEntity(String.class));
}
```

结果结果 ：**20161217,20161218,20161219.**

如果要使用用逗号分开的表示法，您可以添加：

```
public static class MultiValuedParamConverterProvider implements ParamConverterProvider
  @SuppressWarnings("unchecked")
  @Override
  public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation[]
annotations) {
    if (List.class.isAssignableFrom(rawType)) {
      return (ParamConverter<T>) new MultiValuedParamConverter();
    }
    return null;
  }
}

public static class MultiValuedParamConverter implements ParamConverter<List<?>> {
  @Override
  public List<?> fromString(String param) {
    if (param == null || param.trim().isEmpty()) {
      return null;
    }
    return parse(param.split(","));
  }

  @Override
  public String toString(List<?> list) {
    if (list == null || list.isEmpty()) {
      return null;
    }
    return stringify(list);
  }

  private static List<String> parse(String[] params) {
    List<String> list = new ArrayList<String>();
    for (String param : params) {
      list.add(param);
    }
    return list;
  }
}
```

现在，您可以按如下方式调用 **TestResource**:

```
@Test
public void testQueryParamCustom() throws Exception {
  ResteasyClient client = new ResteasyClientBuilder().build();
  Invocation.Builder request = client.target("http://localhost:8081/queryParam?
q=20161217,20161218,20161219").request();
  Response response = request.get();
  System.out.println("response: " + response.readEntity(String.class));
}
```

获得 **回复：20161217,20161218,20161219.**

在本例中，**MultiValuedParamConverter.fromString（）** 函数会创建并返回 **aArrayList**，以便可以重写 **TestResource.conversion（）** 功能：

```
@Path("queryParam")
public static class TestResource {

  @GET
  @Path("")
  public Response conversion(@QueryParam("q") ArrayList<String> list) {
    return Response.ok(stringify(list)).build();
  }
}
```

另外，还可重写 **MultiValuedParamConverter** 以返回 **LinkList**，**TestResource.conversion（）中的**参数列表可以是 **List** 或 **LinkedList**。

最后，请注意，此扩展也适用于数组。例如，

```
public static class Foo {
   private String foo;
   public Foo(String foo) {
      this.foo = foo;
   }
   public String getFoo() {
      return foo;
   }
}

public static class FooArrayParamConverter implements ParamConverter < Foo[] > {
   @Override
   public Foo[] fromString(String value) {
      String[] ss = value.split(",");
      Foo[] fs = new Foo[ss.length];
      int i = 0;
      for (String s: ss) {
         fs[i++] = new Foo(s);
      }
      return fs;
   }

   @Override
   public String toString(Foo[] values) {
      StringBuffer sb = new StringBuffer();
      for (int i = 0; i < values.length; i++) {
```

```
                sb.append(values[i].getFoo()).append(",");
            }
            if (sb.length() > 0) {
                sb.deleteCharAt(sb.length() - 1);
            }
            return sb.toString();
        }
    }

    @Provider
    public static class FooArrayParamConverterProvider implements ParamConverterProvider {
        @SuppressWarnings("unchecked")
        @Override
        public < T > ParamConverter < T > getConverter(Class < T > rawType, Type genericType,
    Annotation[] annotations) {
            if (rawType.equals(Foo[].class));
            return (ParamConverter < T > ) new FooArrayParamConverter();
        }
    }

    @Path("")
    public static class ParamConverterResource {

        @GET
        @Path("test")
        public Response test(@QueryParam("foos") Foo[] foos) {
            return Response.ok(new FooArrayParamConverter().toString(foos)).build();
        }
    }
```

java.util.Optional Parameter Types

RESTEasy 提供了多种额外的 **java.util.Optional** 参数类型。这些参数类型充当打包程序对象类型。它们允许用户输入可选类型参数，并使用 **Optional.orElse（）** 等方法取消所有 null 检查。

```
@Path("/double")
@GET
public String optDouble(@QueryParam("value") OptionalDouble value) {
    return Double.toString(value.orElse(4242.0));
}
```

上例演示了 **OptionalDouble** 可用作参数类型。如果 **@QueryParam** 中未提供值，则将返回默认值。以下参数类型支持可选参数：

- **@QueryParam**
- **@MatrixParam**
- **@FormParam**
- **@HeaderParam**
- **@CookieParam**

2.7.7. Serializable Provider

从不受信任的来源对 Java 对象进行序列化是不安全的。因此，默认情况下会禁用 **org.jboss.resteasy.plugins.providers.SerializableProvider**。不建议使用此提供程序。

2.7.8. JSON 提供者

2.7.8.1. RESTEasy Jackson2 中的 JsonFilter 支持

JsonFilter 允许您为类添加 **@JsonFilter** 注释，从而促进动态过滤。以下示例定义了从 **nameFilter** 类映射到过滤器实例，然后在对实例序列化到 JSON 格式时过滤 bean 属性。

```
@JsonFilter(value="nameFilter")
public class Jackson2Product {
   protected String name;
   protected int id;
   public Jackson2Product() {
   }
   public Jackson2Product(final int id, final String name) {
      this.id = id;
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
}
```

@JsonFilter 注释资源组，以过滤出不应在 JSON 响应中序列化的属性。若要映射过滤器 ID 和实例，您必须创建另一个 Jackson 类，并将 ID 和过滤实例映射到此类，如下例中所示。

```
public class ObjectFilterModifier extends ObjectWriterModifier {
 public ObjectFilterModifier() {
 }
 @Override
 public ObjectWriter modify(EndpointConfigBase<?> endpoint,
  MultivaluedMap<String, Object> httpHeaders, Object valueToWrite,
  ObjectWriter w, JsonGenerator jg) throws IOException {

  FilterProvider filterProvider = new SimpleFilterProvider().addFilter(
   "nameFilter",
   SimpleBeanPropertyFilter.filterOutAllExcept("name"));
  return w.with(filterProvider);
 }
}
```

在上面的示例中，方法 mod （） 负责过滤除 **name** 属性外的所有属性，然后再写入响应。要实现此目的，RESTEasy 必须了解此映射信息。您可以在 **WriterInterceptor** 或 servlet 过滤器中设置映射信息，如下例所示。

示例：使用 WriterInterceptor 设置 ObjectFilterModifier

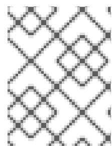
```
@Provider
public class JsonFilterWriteInterceptor implements WriterInterceptor{

 private ObjectFilterModifier modifier = new ObjectFilterModifier();
 @Override
 public void aroundWriteTo(WriterInterceptorContext context)
  throws IOException, WebApplicationException {
     //set a threadlocal modifier
          ObjectWriterInjector.set(modifier);
             context.proceed();
 }

}
```

示例：使用 Servlet Filter 设置 ObjectFilterModifier

```
public class ObjectWriterModifierFilter implements Filter {
 private static ObjectFilterModifier modifier = new ObjectFilterModifier();

 @Override
 public void init(FilterConfig filterConfig) throws ServletException {
 }

 @Override
 public void doFilter(ServletRequest request, ServletResponse response,
  FilterChain chain) throws IOException, ServletException {
     ObjectWriterInjector.set(modifier);
     chain.doFilter(request, response);
 }

 @Override
 public void destroy() {
 }

}
```

现在，RESTEasy 可以从 **ThreadLocal** 变量获取 **ObjectFilterModifier**，并在写入响应前将其配置为修改 **ObjectWriter**。

2.7.8.2. JSON 序列化和持续时间对象



注意

使用 RESTEasy 来序列化时间和持续时间对象，如 **LocalDateTime**、**LocalDate** 或 **Duration** 类，而无需配置 Jackson2 供应商会导致错误。

RESTEasy 支持使用 Jackson2 供应商的序列化时间和持续时间对象。以下示例演示了如何配置 Jackson2 供应商来激活序列化：

示例：为序列化配置 Jackson2 供应商

```
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.ext.ContextResolver;
import javax.ws.rs.ext.Provider;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.json.JsonMapper;

@Provider
@Produces(MediaType.APPLICATION_JSON)
public class JacksonDatatypeJacksonProducer implements ContextResolver<ObjectMapper> {

    private final ObjectMapper json;

    public JacksonDatatypeJacksonProducer() throws Exception {
        this.json = JsonMapper.builder()
                .findAndAddModules()
                .build();
    }

    @Override
    public ObjectMapper getContext(Class<?> objectType) {
        return json;
    }
}
```

2.7.8.3. JSON Binding

RESTEasy 同时支持 Jakarta JSON Binding 和 Jakarta JSON 处理。根据规范，jakarta JSON Binding 的实体提供商优先于所有类型的实体（ **JsonValue** 及其子类型除外）的 Jakarta JSON Processing。

resteasy-json-binding-provider 模块的 **JsonBindingProvider** 属性为 Jakarta JSON Binding 提供支持。为了满足 Jakarta RESTful Web Services 2.1 要求，**JsonBindingProvider** 提供商优先于其他供应商来处理 JSON 载荷，特别是 Jackson 有效负载。

对于相同的输入，Jackson 和 Jakarta JSON Binding 参考实施中的 JSON 输出可能会有所不同。因此，为了保持向后兼容性，您可以将 **resteasy.preferJacksonOverJsonB** 上下文属性设置为 **true**，并为当前部署禁用 **JsonBindingProvider** 配置。

JBoss EAP 支持通过设置同名的系统属性来为 **resteasy.preferJacksonOverJsonB** 上下文属性指定默认值。如果没有为上下文和系统属性设置值，它将扫描 Jackson 注解的 Jakarta RESTful Web Services 部署，并在找到任何这些注解时将属性设置为 **true**。

2.7.9. Jakarta XML Binding Providers

2.7.9.1. Jakarta XML Binding 和 XML Provider

RESTEasy 为 XML 提供 Jakarta XML Binding 供应商支持。

@XmlHeader 和 @Stylesheet
RESTEasy 提供使用 **@org.jboss.resteasy.annotations.providers.jaxb.XmlHeader** 注释设置 XML 标头。

示例：使用 @XmlHeader 注解

```
@XmlRootElement
public static class Thing {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}

@Path("/test")
public static class TestService {

  @GET
  @Path("/header")
  @Produces("application/xml")
  @XmlHeader("<?xml-stylesheet type='text/xsl' href='${baseuri}foo.xsl' ?>")
  public Thing get() {
    Thing thing = new Thing();
    thing.setName("bill");
    return thing;
  }
}
```

@XmlHeader 确保 XML 输出具有 XML 样式表标头。

RESTEasy 具有方便的样式表标头注释。

示例：使用 @Stylesheet 注解

```
@XmlRootElement
public static class Thing {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}

@Path("/test")
public static class TestService {

  @GET
  @Path("/stylesheet")
  @Produces("application/xml")
  @Stylesheet(type="text/css", href="${basepath}foo.xsl")
  @Junk
  public Thing getStyle() {
```

```
        Thing thing = new Thing();
        thing.setName("bill");
        return thing;
    }
}
```

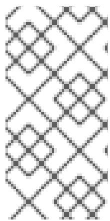
2.7.9.2. Jakarta XML Binding 和 JSON Provider

RESTEasy 允许您使用 JSON 提供程序将 Jakarta XML 绑定标注为 POJO 并从 JSON 中注释。此提供程序打包了 Jackson JSON 库以完成此任务。它具有基于 Java Bans 的模型和 API，类似于 Jakarta XML Binding。

Jackson 已包含 Jakarta RESTful Web 服务集成，但它已由 RESTEasy 扩展。若要将它包含在您的项目中，您需要更新 Maven 依赖项。

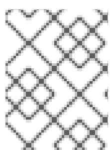
Jackson 的 Maven 依赖项

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson2-provider</artifactId>
    <version>${version.org.jboss.resteasy}</version>
    <scope>provided</scope>
</dependency>
```



注意

RESTEasy 的默认 JSON 提供程序是 Jackson2。早期版本的 JBoss EAP 包含 Jackson1 JSON 提供程序。有关将现有应用从 Jackson1 提供商迁移的更多详细信息，请参阅 JBoss EAP *迁移指南*。如果您仍然要使用 Jackson1 提供程序，您必须 明确更新 Maven 依赖项来获取它。



注意

以前版本的 JBoss EAP 中 RESTEasy 的默认 JSON 提供程序是 Jettison，但现已在 JBoss EAP 7 中弃用。如需了解更多详细信息，请参阅 JBoss EAP *迁移指南*。

JSON 提供程序示例

```
@XmlRootElement
public static class Thing {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}

@Path("/test")
public static class TestService {
```

```
  @GET
  @Path("/thing")
  @Produces("application/json")
  public Thing get() {
    Thing thing = new Thing();
    thing.setName("the thing");
    return thing;
  }
}
```

2.7.9.2.1. Java 8 的 Jackson 模块支持

本节提供 Maven 依赖项，并演示如何在核心 Jackson 模块不需要 Java 8 运行时环境时注册支持 Java 8 功能所需的 Jackson 模块。Jackson 模块包括：

- Java 8 数据类型
- Java 8 日期/时间

添加以下 Maven 依赖项：

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jdk8</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
</dependency>
```

您可以使用 **findAndRegisterModules（）** 或 **ObjectMapper.registerModule（）** 找到并注册所有模块，如下例所示：

```
ObjectMapper mapper = new ObjectMapper();
mapper.findAndRegisterModules();
```

```
ObjectMapper mapper = new ObjectMapper()
   .registerModule(new ParameterNamesModule())
   .registerModule(new Jdk8Module())
   .registerModule(new JavaTimeModule());
```

示例：持续时间数据类型

```
@GET
@Path("/duration")
@Produces(MediaType.APPLICATION_JSON)
public Duration getDuration() {
    return Duration.ofSeconds(5, 6);
}
```

示例：可选数据类型

```
@GET
```

```
@Path("/optional/{nullParam}")
@Produces(MediaType.APPLICATION_JSON)
public Optional<String> getOptional(@PathParam("nullParam") boolean nullParameter) {
   return nullParameter ? Optional.<String>empty() : Optional.of("info@example.com");
}
```

您必须使用 **ContextResolver** 的自定义实施，才能在 RESTEasy 中使用这些 Jackson 模块。

```
@Provider
@Produces(MediaType.APPLICATION_JSON)
public class JacksonDatatypeJacksonProducer implements ContextResolver<ObjectMapper> {
   private final ObjectMapper json;
   public JacksonDatatypeJacksonProducer() throws Exception {
      this.json = new ObjectMapper()
            .findAndRegisterModules()
            .configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false)
            .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
   }
   @Override
   public ObjectMapper getContext(Class<?> objectType) {
      return json;
   }
}
```

2.7.9.2.2. 切换默认的 Jackson 提供程序

JBoss EAP 7 包含 Jackson 2.6.x 或更高版本，以及 **resteasy-jackson2-provider**，现在是默认的 Jackson 提供商。

若要切换到 JBoss EAP 上一发行版中包含的默认 **resteasy-jackson-provider**，请排除新提供程序并在 **jboss-deployment-structure.xml** 应用部署描述符文件中添加上述提供程序的依赖性。

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
  <deployment>
    <exclusions>
      <module name="org.jboss.resteasy.resteasy-jackson2-provider"/>
    </exclusions>
    <dependencies>
      <module name="org.jboss.resteasy.resteasy-jackson-provider" services="import"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

2.7.10. 创建 Jakarta XML Binding Decorators

RESTEasy 的 Jakarta XML Binding 供应商提供了一种可插拔的方式来解码实例和 Unmarshaller 实例。您可以创建一个注解来触发某一实例或 Unmarshaller 实例，这可用于解码方法。

使用 RESTEasy 创建 Jakarta XML Binding Decorator

1. 创建 Processor 类。

   a. 创建一个实现 **DecoratorProcessor<Target, Annotation>** 的类。目标是 Jakarta XML Binding¼er 或 Unmarshaller 类。该注释在第 2 步中创建。

b. 给类标上 **@DecorateTypes，再** 声明解码器应解码的 MIME 类型。

c. 在解码函数中设置属性或值。

示例： Processor Class

```
import org.jboss.resteasy.core.interception.DecoratorProcessor;
import org.jboss.resteasy.annotations.DecorateTypes;
import javax.xml.bind.Marshaller;
import javax.xml.bind.PropertyException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
import java.lang.annotation.Annotation;

@DecorateTypes({"text/*+xml", "application/*+xml"})
public class PrettyProcessor implements DecoratorProcessor<Marshaller, Pretty> {
   public Marshaller decorate(Marshaller target, Pretty annotation,
     Class type, Annotation[] annotations, MediaType mediaType) {
   target.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
   }
}
```

2. 创建注解。

a. 创建一个标有 **@Decorator** 注释的自定义接口。

b. 声明 **@Decorator** 注释的处理器和目标。处理器在第 1 步中创建。目标是 Jakarta XML Binding¼ **er** 或 **Unmarshaller** 类。

示例：带有 @Decorator 注解的自定义接口

```
import org.jboss.resteasy.annotations.Decorator;

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER,
ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Decorator(processor = PrettyProcessor.class, target = Marshaller.class)
public @interface Pretty {}
```

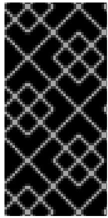
3. 将第 2 步中创建的注释添加到一个函数，以便在处理时对输入或输出进行解码。

您现在已创建了 Jakarta XML Binding decorator，可以在 Jakarta RESTful Web 服务中应用。

2.7.11. Jakarta RESTful Web 服务的多部分提供者

多部分 MIME 格式用于传递嵌入在同一消息中的内容正文列表。多部分 MIME 格式的一个示例是 **multipart/form-data** MIME 类型。这通常在 Web 应用 HTML 表单文档中找到，通常用于上传文件。此 MIME 类型的格式 **数据** 格式与其他多部分格式相同，不同之处在于每个内嵌的内容都有相关联的名称。

RESTEasy 允许 **multipart/form-data** 和 **multipart/*** MIME 类型。RESTEasy 还提供了自定义 API，用于读取和写入多部分类型，以及托管任意 **列表** （任何多部分类型）和 **Map** （仅限多部件/数据）对象。



重要

有许多框架通过过滤器和拦截器（如 Seam 中的 **org.jboss.seam.web.MultipartFilter**）和 Spring 中的 **org.springframework.web.multipart.MultipartResolver** 自动进行多部分解析。但是，传入的多部分请求流只能被解析一次。使用多部分的 RESTEasy 用户应确保在 RESTEasy 获取流之前不会解析流。

2.7.11.1. 多部分数据输入

在编写 Jakarta RESTful Web 服务时，RESTEasy 提供 **org.jboss.resteasy.plugins.providers.multipart.MultipartInput** 接口，以便您可以在任何多部分 MIME 类型中进行读取。

```
package org.jboss.resteasy.plugins.providers.multipart;

public interface MultipartInput {

   List<InputPart> getParts();
   String getPreamble();

   // You must call close to delete any temporary files created
   // Otherwise they will be deleted on garbage collection or on JVM exit
   void close();
}

public interface InputPart {

   MultivaluedMap<String, String> getHeaders();
   String getBodyAsString();
   <T> T getBody(Class<T> type, Type genericType) throws IOException;
   <T> T getBody(org.jboss.resteasy.util.GenericType<T> type) throws IOException;
   MediaType getMediaType();
   boolean isContentTypeFromMessage();
}
```

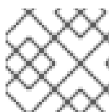
MultipartInput 是一个简单界面，可让您访问多部分消息的每个部分。每个部分都由一个 **InputPart** 接口表示，每个部分都有一组与其关联的标头。您可以通过调用其中一个 **getBody（）** 方法来解译该部分。**genericType** 参数可以是 **null**，但必须设置 **type** 参数。RESTEasy 将根据部分的媒体类型以及您传递的类型信息查找 **MessageBodyReader**。

2.7.11.1.1. 多部分/混合输入

示例：Unmarshalling 部分

```
@Path("/multipart")
public class MyService {

    @PUT
    @Consumes("multipart/mixed")
    public void put(MultipartInput input) {
        List<Customer> customers = new ArrayList...;
        for (InputPart part : input.getParts()) {
            Customer cust = part.getBody(Customer.class, null);
            customers.add(cust);
        }
```

```
        input.close();
    }
}
```



注意

上例假定 **客户** 类标有 Jakarta XML Binding。

有时，您可能想要解译对通用类型的元数据敏感的正文部分。在这种情况下，您可以使用 **org.jboss.resteasy.util.GenericType** 类。

示例：将 Type Sensitive Unmarshalling a Generic Type Metadata

```
@Path("/multipart")
public class MyService {

    @PUT
    @Consumes("multipart/mixed")
    public void put(MultipartInput input) {
        for (InputPart part : input.getParts()) {
            List<Customer> cust = part.getBody(new GenericType<List<Customer>>() {});
        }
        input.close();
    }
}
```

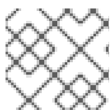
需要使用 **GenericType**，因为它是在运行时获取通用类型信息的唯一方法。

2.7.11.1.2. 多部分/mixed 和 java.util.List输入

如果正文部分是统一的，您不必手动解译每个部分。您只需提供 **java.util.List** 作为您的输入参数。它的类型必须与 **List** 类型声明的 generic 参数解开。

示例：解压 客户列表

```
@Path("/multipart")
public class MyService {

    @PUT
    @Consumes("multipart/mixed")
    public void put(List<Customer> customers) {
        ...
    }
}
```



注意

上例假定 **客户** 类标有 Jakarta XML Binding。

2.7.11.1.3. 使用 multipart/form-data输入

在编写 Jakarta RESTful Web 服务时，RESTEasy 提供了一个界面，允许您在 **多部件/格式数据** MIME 类型中读取。**多部分/格式数据** 通常在 Web 应用 HTML 表单文档中找到，通常用于上传文件。 **form-data**

格式与其他多部分格式相同，不同之处在于每个内嵌的内容都有相关联的名称。用于表单数据输入的界面为 **org.jboss.resteasy.plugins.providers.multipart.MultipartFormDataInput**。

示例： MultipartFormDataInput Interface

```
public interface MultipartFormDataInput extends MultipartInput {

    @Deprecated
    Map<String, InputPart> getFormData();
    Map<String, List<InputPart>> getFormDataMap();
    <T> T getFormDataPart(String key, Class<T> rawType, Type genericType) throws IOException;
    <T> T getFormDataPart(String key, GenericType<T> type) throws IOException;
}
```

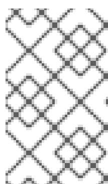
它的工作方式与 前面描述的 **MultipartInput** 基本相同。

2.7.11.1.4. 带有 multipart/form-data的 Java.util.Map

使用表单数据时，如果正文部分是统一的，您不必手动分隔每个部分和每个部分。您只需将 **java.util.Map** 作为输入参数提供。它的类型必须与 **List** 类型声明的 generic 参数解开。

示例：解压 客户 对象的映射

```
@Path("/multipart")
public class MyService {

    @PUT
    @Consumes("multipart/form-data")
    public void put(Map<String, Customer> customers) {
        ...
    }
}
```



注意

上例假定 客户 类标有 Jakarta XML Binding。

2.7.11.1.5. 多部分/相关输入

在编写 Jakarta RESTful Web 服务时，RESTEasy 提供了一个界面，允许您在 多部分/相关 MIME 类型中进行读取。多部分/相关 用于表示不应单独考虑消息部分，而是作为整个聚合的一部分，由 RFC 2387 定义。

多部分/相关的 示例用法是发送带有单条消息中的映像的网页。每个 多部分/相关 消息都有一个 root/start 部分，该部分引用邮件的其他部分。这些部分通过其 Content-ID 标头标识。用于相关输入的界面为 org.jboss.resteasy.plugins.providers.multipart.MultipartRelatedInput。

示例： MultipartRelatedInput Interface

```
public interface MultipartRelatedInput extends MultipartInput {

   String getType();
   String getStart();
   String getStartInfo();
   InputPart getRootPart();
   Map<String, InputPart> getRelatedMap();
}
```

它的工作方式与 MultipartInput 大致相同。

2.7.11.2. 使用多部分数据的输出

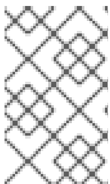
RESTEasy 提供了一个简单的 API 来输出多部分数据。

```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartOutput {

   public OutputPart addPart(Object entity, MediaType mediaType)
   public OutputPart addPart(Object entity, GenericType type, MediaType mediaType)
   public OutputPart addPart(Object entity, Class type, Type genericType, MediaType
mediaType)
   public List<OutputPart> getParts()
   public String getBoundary()
   public void setBoundary(String boundary)
}

public class OutputPart {

   public MultivaluedMap<String, Object> getHeaders()
   public Object getEntity()
   public Class getType()
   public Type getGenericType()
   public MediaType getMediaType()
}
```

若要输出多部分数据，您需要创建一个 MultipartOutput 对象并调用 addPart（） 方法。RESTEasy 将自动查找 MessageBodyWriter 来托管您的实体对象。与 MultipartInput 类似，有时您可能会有对通用类型的元数据敏感的托管。在这种情况下，使用 GenericType。通常，传递对象及其 MediaType 应该足够。

示例：返回 多部分/混合 格式

```
@Path("/multipart")
public class MyService {

   @GET
   @Produces("multipart/mixed")
   public MultipartOutput get() {

      MultipartOutput output = new MultipartOutput();
      output.addPart(new Customer("bill"), MediaType.APPLICATION_XML_TYPE);
      output.addPart(new Customer("monica"), MediaType.APPLICATION_XML_TYPE);
      return output;
   }
}
```



注意

上例假定 客户 类标有 Jakarta XML Binding。

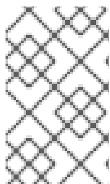
2.7.11.2.1. 使用 java.util.List的多部分输出

如果正文部分是统一的，您不必手动托管每个部分，甚至每个部分，甚至使用 MultipartOutput 对象。您可以提供 java.util.List，该类型必须具有正在捆绑的通用类型，并附带 List 类型声明的 generic 参数。您还必须使用 @PartType 注释给方法标注，以指定各个部分的媒体类型。

示例：返回 客户 对象 列表

```
@Path("/multipart")
public class MyService {

   @GET
   @Produces("multipart/mixed")
   @PartType("application/xml")
```

```
    public List<Customer> get(){
      ...
    }
}
```



注意

上例假定 客户 类标有 Jakarta XML Binding。

2.7.11.2.2. 带有 multipart/form-data的输出

RESTEasy 提供了一个简单的 API 来输出 multipart/form-data。

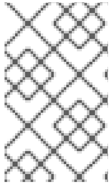
```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartFormDataOutput extends MultipartOutput {

    public OutputPart addFormData(String key, Object entity, MediaType mediaType)
    public OutputPart addFormData(String key, Object entity, GenericType type, MediaType
mediaType)
    public OutputPart addFormData(String key, Object entity, Class type, Type genericType,
MediaType mediaType)
    public Map<String, OutputPart> getFormData()
}
```

要输出 multipart/form-data，您必须创建一个 MultipartFormDataOutput 对象并调用 addFormData（） 方法。RESTEasy 将自动查找 MessageBodyWriter 来托管您的实体对象。与 MultipartInput 类似，有时您可能会有对通用类型的元数据敏感的托管。在这种情况下，使用 GenericType。通常，传递对象及其 MediaType 应该足够。

示例：返回 multipart/form-data Format

```
@Path("/form")
public class MyService {

    @GET
    @Produces("multipart/form-data")
    public MultipartFormDataOutput get() {

        MultipartFormDataOutput output = new MultipartFormDataOutput();
        output.addPart("bill", new Customer("bill"), MediaType.APPLICATION_XML_TYPE);
        output.addPart("monica", new Customer("monica"),
```

```
MediaType.APPLICATION_XML_TYPE);
      return output;
   }
}
```



注意

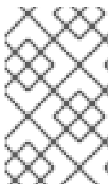
上例假定 客户 类标有 Jakarta XML Binding。

2.7.11.2.3. 使用 java.util.Map的 Multipart FormData Output

如果正文部分是统一的，您不必手动托管每个部分或使用 MultipartFormDataOutput 对象。您只需提供一个 java.util.Map，该类型必须具有正在使用 Map 类型声明的 generic 参数的通用类型。您还必须使用 @PartType 注释给方法标注，以指定各个部分的媒体类型。

示例：返回 客户 对象的映射

```
@Path("/multipart")
public class MyService {

   @GET
   @Produces("multipart/form-data")
   @PartType("application/xml")
   public Map<String, Customer> get() {
      ...
   }
}
```



注意

上例假定 客户 类标有 Jakarta XML Binding。

2.7.11.2.4. 带有 多部分/相关输出

RESTEasy 提供了一个简单的 API，用于输出 多部分/相关。

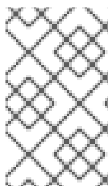
■

```
package org.jboss.resteasy.plugins.providers.multipart;

public class MultipartRelatedOutput extends MultipartOutput {

   public OutputPart getRootPart()
   public OutputPart addPart(Object entity, MediaType mediaType,
      String contentId, String contentTransferEncoding)
   public String getStartInfo()
   public void setStartInfo(String startInfo)
}
```

要输出 multipart/ 相关，您必须创建一个 MultipartRelatedOutput 对象并调用 addPart（） 方法。添加的第一个部分用作 多部分/相关消息的根部分，RESTEasy 会自动查找 MessageBodyWriter 来托管您的实体对象。与 MultipartInput 类似，有时您可能会有对通用类型的元数据敏感的托管。在这种情况下，使用 GenericType。通常，传递对象及其 MediaType 应该足够。

示例：返回 多部分/相关 格式以发送两个镜像

```
@Path("/related")
public class MyService {

   @GET
   @Produces("multipart/related")
   public MultipartRelatedOutput get() {

      MultipartRelatedOutput output = new MultipartRelatedOutput();
      output.setStartInfo("text/html");

      Map<String, String> mediaTypeParameters = new LinkedHashMap<String, String>();
      mediaTypeParameters.put("charset", "UTF-8");
      mediaTypeParameters.put("type", "text/html");
      output.addPart(
         "<html><body>\n"
         + "This is me: <img src='cid:http://example.org/me.png' />\n"
         + "<br />This is you: <img src='cid:http://example.org/you.png' />\n"
         + "</body></html>",
         new MediaType("text", "html", mediaTypeParameters),
         "<mymessage.xml@example.org>", "8bit");
      output.addPart("// binary octets for me png",
         new MediaType("image", "png"), "<http://example.org/me.png>",
         "binary");
      output.addPart("// binary octets for you png", new MediaType(
         "image", "png"),
         "<http://example.org/you.png>", "binary");
      client.putRelated(output);
      return output;
   }
}
```



注意

上例假定 客户 类标有 Jakarta XML Binding。

2.7.11.3. 将多部分表单映射到 POJO

如果您对多部分/格式数据数据包有精确的了解，您可以将它们映射到 POJO 类或从 POJO 类中进行映射。这通过 org.jboss.resteasy.annotations.providers.multipart.MultipartForm 注释 (@MultipartForm)和 Jakarta RESTful Web Services @FormParam 注释来实现。为此，您需要使用至少一个默认构造器定义 POJO，并使用 @FormParams 给其字段和/或属性标注。如果要创建输出，还必须使用 org.jboss.resteasy.annotations.providers.multipart.PartType(@PartType)标记这些 @FormParam s。

示例：将多组件表单映射到 POJO

```
public class CustomerProblemForm {

   @FormParam("customer")
   @PartType("application/xml")
   private Customer customer;

   @FormParam("problem")
   @PartType("text/plain")
   private String problem;

   public Customer getCustomer() { return customer; }
   public void setCustomer(Customer cust) { this.customer = cust; }
   public String getProblem() { return problem; }
   public void setProblem(String problem) { this.problem = problem; }
}
```

在定义了 POJO 类后，您可以使用它来表示 多部分/格式数据。

示例：提交 CustomerProblemForm

```
@Path("portal")
```

```
public interface CustomerPortal {

   @Path("issues/{id}")
   @Consumes("multipart/form-data")
   @PUT
   public void putProblem(@MultipartForm CustomerProblemForm,
                     @PathParam("id") int id);
}

// Somewhere using it:
{
   CustomerPortal portal = ProxyFactory.create(CustomerPortal.class, "http://example.com");
   CustomerProblemForm form = new CustomerProblemForm();
   form.setCustomer(...);
   form.setProblem(...);

   portal.putProblem(form, 333);
}
```

@MultipartForm 注释告知 RESTEasy 对象具有 @FormParam，并且应当从中托管。您还可以使用相同的对象来接收多部分数据。

示例：接收 客户ProblemForm

```
@Path("portal")
public class CustomerPortalServer {

   @Path("issues/{id})
   @Consumes("multipart/form-data")
   @PUT
   public void putIssue(@MultipartForm CustomerProblemForm,
                     @PathParam("id") int id) {
     ... write to database...
   }
}
```

2.7.11.4. XML-二进制优化打包(XOP)

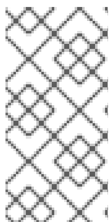
如果您有一个 Jakarta XML Binding 标注了 POJO，并且也包含一些二进制内容，您可以选择将其发送成不需要以任何方式（如 base64 或十六进制）对其进行编码。这可通过 XOP 实现，从而在仍然使用方便的 POJO 的同时加快传输速度。

RESTEasy 允许将 XOP 消息打包为 多部分/相关.

要配置 XOP，您首先需要一个带注解的 POJO 的 Jakarta XML Binding。

示例：jakarta XML Binding POJO

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Xop {

   private Customer bill;
   private Customer monica;

   @XmlMimeType(MediaType.APPLICATION_OCTET_STREAM)
   private byte[] myBinary;

   @XmlMimeType(MediaType.APPLICATION_OCTET_STREAM)
   private DataHandler myDataHandler;

   // methods, other fields ...
}
```



注意

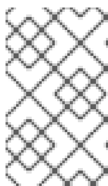
@XmlMimeType 告知 Jakarta XML 绑定二进制内容的 mime 类型。这不需要执行 XOP 打包，但如果您知道确切的类型，则建议对其进行设置。

在上面的 POJO myBinary 和 myDataHandler 中，将处理为二进制附件，而整个 XOP 对象将作为 XML 发送。代替二进制文件，将仅生成其引用。javax.activation.DataHandler 是最常用的类型。如果需要 java.io.InputStream 或 javax.activation.DataSource，您需要使用 DataHandler。Java.awt.Image 和 javax.xml.transform.SourceSome 也可用。

示例：使用 XOP 的客户端发送二进制内容

```
// our client interface:
@Path("mime")
public static interface MultipartClient {
```

```
    @Path("xop")
    @PUT
    @Consumes(MultipartConstants.MULTIPART_RELATED)
    public void putXop(@XopWithMultipartRelated Xop bean);
}

// Somewhere using it:
{
    MultipartClient client = ProxyFactory.create(MultipartClient.class,
        "http://www.example.org");
    Xop xop = new Xop(new Customer("bill"), new Customer("monica"),
        "Hello Xop World!".getBytes("UTF-8"),
        new DataHandler(new ByteArrayDataSource("Hello Xop World!".getBytes("UTF-8"),
        MediaType.APPLICATION_OCTET_STREAM)));
    client.putXop(xop);
}
```



注意

上例假定 客户 类标有 Jakarta XML Binding。

@Consumes(MultipartConstants.MULTIPART_RELATED) 用于告知 RESTEasy，您要发送 多部分/相关 软件包，这是保存 XOP 消息的容器格式。@XopWithMultipartRelated 用于告知 RESTEasy，您想要发出 XOP 消息。

示例：用于接收 XOP 的 RESTEasy Server

```
@Path("/mime")
public class XopService {
    @PUT
    @Path("xop")
    @Consumes(MultipartConstants.MULTIPART_RELATED)
    public void putXopWithMultipartRelated(@XopWithMultipartRelated Xop xop) {
        // do very important things here
    }
}
```

@consumes(MultipartConstants.MULTIPART_RELATED) 用于告知 RESTEasy，您希望读取 多部分/相关 包。@XopWithMultipartRelated 用于 告知 RESTEasy 想要读取 XOP 消息。您可以将

RESTEasy 服务器配置为以类似的方式生成 XOP 值，方法是添加 @Produces 注释并返回适当的类型。

2.7.11.5. 多部分消息覆盖默认回退内容类型

默认情况下，如果部分 文本/plain 中不存在 Content-Type 标头；charset=us-ascii 用作回退。这由 MIME RFC 定义。但是，某些 Web 客户端（如许多浏览器）可能会为文件部分发送 Content-Type 标头，但不适用于 多部分/格式数据 请求中的所有字段。这可能会导致服务器端的字符编码和取消处理错误。RESTEasy 的 PreProcessInterceptor 基础架构可用于更正此问题。您可以使用它为每个请求动态定义另一个非 RFC 兼容回退值。

示例：将 * / *; charset=UTF-8 设置为默认 Fallback

```
import org.jboss.resteasy.plugins.providers.multipart.InputPart;

@Provider
@ServerInterceptor
public class ContentTypeSetterPreProcessorInterceptor implements PreProcessInterceptor {

   public ServerResponse preProcess(HttpRequest request, ResourceMethod method)
         throws Failure, WebApplicationException {
      request.setAttribute(InputPart.DEFAULT_CONTENT_TYPE_PROPERTY, "*/*;
charset=UTF-8");
      return null;
   }
}
```

2.7.11.6. 多部分消息的内容类型覆盖

通过使用拦截器和 InputPart.DEFAULT_CONTENT_TYPE_PROPERTY 属性，您可以设置默认的 Content-Type。您还可以通过调用 org.jboss.resteasy.plugins.providers.multipart.InputPart.setMediaType（） 在任何输入部分中覆盖 Content- Type。

示例：覆盖 Content-Type

```
@POST
@Path("query")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.TEXT_PLAIN)
public Response setMediaType(MultipartInput input) throws IOException {
```

```
    List<InputPart> parts = input.getParts();
    InputPart part = parts.get(0);
    part.setMediaType(MediaType.valueOf("application/foo+xml"));
    String s = part.getBody(String.class, null);
    ...
}
```

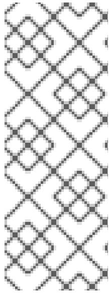
2.7.11.7. 为多部分消息覆盖默认 Fallback charset

在某些情况下，多部分消息的部分可能具有 Content-Type 标头且没有 charset 参数。如果设置了 InputPart.DEFAULT_CONTENT_TYPE_PROPERTY 属性，并且值具有 charset 参数，则该值将被附加到没有 charset 参数的现有 Content-Type 标头中。

您还可以使用 常量 InputPart.DEFAULT_CHARSET_PROPERTY(resteasy.provider.multipart.inputpart.defaultCharset) 指定默认的 char set。

示例：指定默认 charset

```
import org.jboss.resteasy.plugins.providers.multipart.InputPart;

@Provider
@ServerInterceptor
public class ContentTypeSetterPreProcessorInterceptor implements PreProcessInterceptor {

    public ServerResponse preProcess(HttpRequest request, ResourceMethod method)
            throws Failure, WebApplicationException {
        request.setAttribute(InputPart.DEFAULT_CHARSET_PROPERTY, "UTF-8");
        return null;
    }
}
```



注意

如果设置了 InputPart.DEFAULT_CONTENT_TYPE_PROPERTY 和 InputPart.DEFAULT_CHARSET_PROPERTY，则 InputPart.DEFAULT_CHARSET_PROPERTY 的值将覆盖 InputPart.DEFAULT_CONTY 的值中的任何 charset。

2.7.11.8. 使用 RESTEasy 客户端发送多部分实体

除了配置多部分提供程序外，您还可以配置 RESTEasy 客户端来发送多部分数据。

使用 RESTEasy 客户端类

要在应用程序中使用 RESTEasy 客户端类，您必须将 Maven 依赖项添加到项目的 POM 文件中。

示例：Maven 依赖项

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-client</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-multipart-provider</artifactId>
  <version>${version.org.jboss.resteasy}</version>
  <scope>provided</scope>
</dependency>
```

使用 RESTEasy 客户端发送多部分数据

若要发送多部分数据，您必须首先配置 RESTEasy 客户端，并构建一个 org.jboss.resteasy.plugins.providers.multipart.MultipartFormDataOutput 对象来包含您的多部分数据。然后，您可以使用客户端将该 MultipartFormDataOutput 对象作为 javax.ws.rs.core.GenericEntity 发送。

示例：RESTEasy Client

```
ResteasyClient client = new ResteasyClientBuilder().build();
```

```
ResteasyWebTarget target = client.target("http://foo.com/resource");

MultipartFormDataOutput formOutputData = new MultipartFormDataOutput();
formOutputData.addFormData("part1", "this is part 1", MediaType.TEXT_PLAIN);
formOutputData.addFormData("part2", "this is part 2", MediaType.TEXT_PLAIN);

GenericEntity<MultipartFormDataOutput> data = new
GenericEntity<MultipartFormDataOutput>(formOutputData) { };

Response response = target.request().put(Entity.entity(data,
MediaType.MULTIPART_FORM_DATA_TYPE));

response.close();
```

2.7.12. RESTEasy Atom 支持

RESTEasy Atom API 和提供程序是 RESTEasy 定义为代表 Atom 的简单对象模型。API 的主要类位于 org.jboss.resteasy.plugins.providers.atom 软件包中。RESTEasy 使用 Jakarta XML Binding 来托管和卸载 API。提供程序基于 Jakarta XML Binding，不限于使用 XML 发送 Atom 对象。RESTEasy 能够被 Atom API 和供应商（包括 JSON）重复利用的所有 Jakarta XML Binding 提供程序。

```
import org.jboss.resteasy.plugins.providers.atom.Content;
import org.jboss.resteasy.plugins.providers.atom.Entry;
import org.jboss.resteasy.plugins.providers.atom.Feed;
import org.jboss.resteasy.plugins.providers.atom.Link;
import org.jboss.resteasy.plugins.providers.atom.Person;

@Path("atom")
public class MyAtomService {

  @GET
  @Path("feed")
  @Produces("application/atom+xml")
  public Feed getFeed() throws URISyntaxException {
    Feed feed = new Feed();
    feed.setId(new URI("http://example.com/42"));
    feed.setTitle("My Feed");
    feed.setUpdated(new Date());
    Link link = new Link();
    link.setHref(new URI("http://localhost"));
    link.setRel("edit");
    feed.getLinks().add(link);
    feed.getAuthors().add(new Person("John Brown"));
    Entry entry = new Entry();
    entry.setTitle("Hello World");
    Content content = new Content();
    content.setType(MediaType.TEXT_HTML_TYPE);
    content.setText("Nothing much");
    entry.setContent(content);
```

```
      feed.getEntries().add(entry);
      return feed;
   }
}
```

2.7.12.1. 将 Jakarta XML 绑定与 Atom Provider 搭配使用

org.jboss.resteasy.plugins.providers.atom.Content 类允许您解封和 marshal Jakarta XML Binding 注解为内容主体的对象。

示例：与客户进行输入

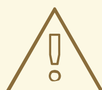
```
@XmlRootElement(namespace = "http://jboss.org/Customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
   @XmlElement
   private String name;

   public Customer() {
   }

   public Customer(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}

@Path("atom")
public static class AtomServer {
   @GET
   @Path("entry")
   @Produces("application/atom+xml")
   public Entry getEntry() {
      Entry entry = new Entry();
      entry.setTitle("Hello World");
      Content content = new Content();
      content.setJAXBObject(new Customer("bill"));
      entry.setContent(content);
      return entry;
   }
}
```

Content.setJAXBObject（） 方法允许您将发送到 Jakarta XML Binding 的内容对象相应地指定为 marshal。如果您使用与 XML 不同的基本格式，即 application/atom+json，附加的 Jakarta XML Binding 对象会以相同的格式 marshalled。如果您有 Atom 文档作为输入，您也可以使用 Content.getJAXBObject(Class clazz) 方法从内容中提取 Jakarta XML Binding 对象。

示例：属性文档提取客户对象

```
@Path("atom")
public static class AtomServer {
  @PUT
  @Path("entry")
  @Produces("application/atom+xml")
  public void putCustomer(Entry entry) {
    Content content = entry.getContent();
    Customer cust = content.getJAXBObject(Customer.class);
  }
}
```

2.7.13. YAML 供应商



警告

不支持 resteasy-yaml-provider 模块。不建议使用，因为 RESTEasy 用于取消托管的 SnakeYAML 库中存在安全问题。

RESTEasy 附带使用 SnakeYAML 库构建的 YAML 支持。

在 JBoss EAP 7.1 之前的版本中，YAML 提供程序设置默认启用，您只需要为 YAML 配置 Maven 依赖项，即可在应用中使用。自 JBoss EAP 7.1 起，默认情况下，YAML 提供程序是禁用的，应用中必须明确启用 YAML 提供程序。

启用 YAML 提供者

要在应用程序中启用 YAML 供应商，请按照以下步骤执行：

1. 创建或更新名为 **javax.ws.rs.ext.Providers** 的文件。

2. 将以下内容添加到 文件中：

```
org.jboss.resteasy.plugins.providers.YamlProvider
```

3. 将文件放到 WAR 或 JAR 文件的 **META-INF/services/** 文件夹中。

YAML Provider Maven 依赖项

要在应用程序中使用 YAML 供应商，您必须将 snakeyaml JAR 依赖项添加到应用程序的项目 POM 文件中。

示例：YAML 的 Maven 依赖项

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-yaml-provider</artifactId>
   <version>${version.org.jboss.resteasy}</version>
   <scope>provided</scope>
</dependency>

<dependency>
   <groupId>org.yaml</groupId>
   <artifactId>snakeyaml</artifactId>
   <version>${version.org.yaml.snakeyaml}</version>
</dependency>
```

YAML 提供程序代码示例

YAML 供应商识别三种 mime 类型：

- **text/x-yaml**

- **text/yaml**

- application/x-yaml

以下是如何在资源方法中使用 YAML 的示例。

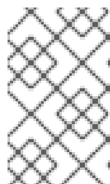
示例：资源生成 YAML

```
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/yaml")
public class YamlResource {

  @GET
  @Produces("text/x-yaml")
  public MyObject getMyObject() {
    return createMyObject();
  }
...
}
```

2.8. 使用 JAKARTA JSON 处理

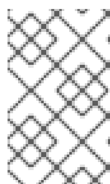
Jakarta JSON 处理在 Jakarta JSON 处理 1.1 规范中定义。

Jakarta JSON 处理定义用于处理 JSON 的 API。JBoss EAP 支持 javax.json.JsonObject、javax.json.JsonArray 和 javax.json.JsonStructure 作为请求或响应实体。



注意

Jakarta JSON 处理与使用 Padding(JSONP)的 JSON 不同。



注意

如果 Jakarta JSON 处理位于同一个类路径上，则不会与 Jackson 冲突。

若要创建 JsonObject，请通过调用 Json.createObjectBuilder（）并构建 JSON 对象来使用 JsonObjectBuilder。

示例：创建 javax.json.JsonObject

```
JsonObject obj = Json.createObjectBuilder().add("name", "Bill").build();
```

示例： javax.json.JsonObject的 Corresponding JSON

```
{
  "name":"Bill"
}
```

若要创建 JsonArray，请通过调用 Json.createArrayBuilder（）并构建 JSON 数组来使用 JsonArrayBuilderBuilder。

示例：创建 javax.json.JsonArray

```
JsonArray array =
  Json.createArrayBuilder()
    .add(Json.createObjectBuilder().add("name", "Bill").build())
    .add(Json.createObjectBuilder().add("name", "Monica").build()).build();
```

示例：用于 javax.json.JsonArray的 Corresponding JSON

```
[
  {
  "name":"Bill"
```

```
  },
  {
  "name":"Monica"
  }
]
```

JsonStructure 是 JsonObject 和Json Array 的父类。

示例：创建 javax.json.JsonStructure

```
JsonObject obj = Json.createObjectBuilder().add("name", "Bill").build();

JsonArray array =
  Json.createArrayBuilder()
    .add(Json.createObjectBuilder().add("name", "Bill").build())
    .add(Json.createObjectBuilder().add("name", "Monica").build()).build();

JsonStructure sObj = (JsonStructure) obj;
JsonStructure sArray = (JsonStructure) array;
```

您可以在 Jakarta RESTful Web Services 资源中直接使用 JsonObject、JsonArray 和 JsonStructure。

示例：使用 Jakarta JSON 处理的 Jakarta RESTful Web 服务资源

```
@Path("object")
@POST
@Produces("application/json")
@Consumes("application/json")
public JsonObject object(JsonObject obj) {
  // do something
  return obj;
 }

@Path("array")
@POST
@Produces("application/json")
@Consumes("application/json")
```

```
public JsonArray array(JsonArray array) {
  // do something
  return array;
}

@Path("structure")
@POST
@Produces("application/json")
@Consumes("application/json")
public JsonStructure structure(JsonStructure structure) {
  // do something
  return structure;
}
```

您还可以使用客户端的 Jakarta JSON Processing 来发送 JSON。

示例：使用 Jakarta JSON 处理的客户端

```
WebTarget target = client.target(...);
JsonObject obj = Json.createObjectBuilder().add("name", "Bill").build();
JsonObject newObj = target.request().post(Entity.json(obj), JsonObject.class);
```

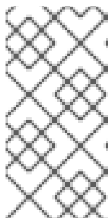
2.9. RESTEASY/JAKARTA ENTERPRISE BEANS 集成

要将 RESTEasy 与 Jakarta 企业 Beans 集成，请将 Jakarta RESTful Web 服务注释添加到您要公开为 Jakarta RESTful Web Services 端点的 Jakarta 企业 Beans 类。您还可以对 bean 的业务界面应用注释。有两种方法可以将 bean 作为端点激活：

- 使用 web.xml 文件：
- 使用 javax.ws.rs.core.Application.

要使 Jakarta 企业 Beans 成为 Jakarta RESTful Web Services 资源，请使用 Jakarta RESTful Web Services 注释给无状态会话 @Remote 或 @Local 接口标注：

```
@Local
@Path("/Library")
public interface Library {
  @GET
  @Path("/books/{isbn}")
  public String getBook(@PathParam("isbn") String isbn);
}
@Stateless
public class LibraryBean implements Library {
...
}
```



注意

请注意，L ibri 接口 使用完全限定名称引用，而 LibraryBean 仅通过简单的类名称来引用。

然后，使用 RESTEasy web.xml 文件中的 resteasy.jndi.resources 上下文参数手动使用 RESTEasy 注册 Jakarta Enterprise Beans：

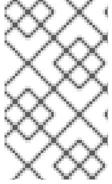
```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <context-param>
    <param-name>resteasy.jndi.resources</param-name>
    <param-value>java:module/LibraryBean!org.app.Library</param-value>
  </context-param>
  <listener>
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

您还可以为 resteasy.jndi.resources 上下文参数指定多个 Java 命名和目录接口名称，即 Jakarta Enterprise Beans（用逗号分开）。

将 Jakarta EE 标准激活为 RESTEasy 端点的 Jakarta EE 标准方法是使用 javax.ws.rs.core.Application。这可以通过将 Jakarta Enterprise Beans 实施类包含在应用的 getClasses（） 方法返回的集合中来实现。此方法不需要在 web.xml 文件中指定任何内容。

如需演示 RESTEasy 与 Jakarta Enterprise Beanss 集成 的 RESTEasy 集成，请参阅 JBoss EAP 附带的 kitchensink、helloworld-html5 和 managed-executor-service 快速入门。

2.10. SPRING 集成



注意

您的应用程序必须具有现有的 Jakarta XML Web 服务服务和客户端配置。

RESTEasy 与 Spring 4.2.x 集成。

Maven 用户必须使用 resteasy-spring 构件。此外，JAR 也作为 JBoss EAP 中的模块提供。

RESTEasy 附带自己的 Spring ContextLoaderListener，其注册了一个特定的 BeanPostProcessor，它在 BeanFactory 创建 BeanFactory 时处理 Jakarta RESTful Web Services 注释。这意味着 RESTEasy 自动扫描 Bean 类上的 @Provider 和 Jakarta RESTful Web Services 资源注释，并将它们注册为 Jakarta RESTful Web Services 资源。

在 web.xml 文件中添加以下内容以启用 RESTEasy/Spring 集成功能：

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <listener>
    <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
  </listener>
  <listener>
    <listener-class>org.jboss.resteasy.plugins.spring.SpringContextLoaderListener</listener-
class>
  </listener>
  <listener>
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-
class>
  </listener>
  <servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Resteasy</servlet-name>
```

```
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

在 ResteasyBootstrap 使用由它初始化的 ServletContext 属性后，必须声明 SpringContext LoaderListener。

有关演示 RESTEasy 与 SpringEasy 集成的 Web 应用的工作示例，请参见 JBoss EAP 附带的 Spring -resteasy 快速入门。

2.11. JAKARTA 上下文和依赖注入集成

RESTEasy 和 Jakarta Contexts 和 Dependency Injection 之间的集成由 resteasy-cdi 模块提供。

Jakarta RESTful Web 服务和 Jakarta 上下文和依赖注入规范都引入了自己的组件模型。置于 Jakarta 上下文和依赖注入存档中的每个类均满足一组基本限制，隐式为 Jakarta 上下文和依赖注入 Ban。为了使 Java 类成为 Jakarta RESTful Web 服务组件，需要使用 @Path 或 @Provider 明确声明您的 Java 类。如果没有集成代码，注解适合使用 Jakarta Contexts 和 Dependency Injection Bjection Bjection Bjection 的类会导致错误结果，并且 Jakarta RESTful Web Services 组件不由 Jakarta Contexts 和 Dependency Injection 管理。resteasy-cdi 模块是一个网桥，它允许 RESTEasy 处理从 Jakarta Contexts 和 Dependency Injection 容器获取的类实例。

在 Web 服务调用期间，resteasy-cdi 模块向 Jakarta Contexts 和 Dependency Injection 容器询问 Jakarta RESTful Web Services 组件的托管实例。然后，此实例将传递到 RESTEasy。如果受管实例因某种原因不可用，例如类放置在非 bean 部署存档的 JAR 文件中，RESTEasy 会返回实例化类本身。

因此，Jakarta 上下文和依赖注入服务（如注入、生命周期管理、事件、解码和拦截器绑定）可用于 Jakarta RESTful Web Services 组件。

2.11.1. 默认范围

默认情况下，未明确定义范围的 Jakarta Contexts 和 Dependency Injection bean。这种伪范围意味着 Bean 能够适应它所注入的 Bean 的生命周期。随着组件的生命周期界限明确指定，常规范围（包括请求、会话和应用程序）更适合 Jakarta RESTful Web 服务组件。因此，resteasy-cdi 模块会以以下方式更改默认范围范围：

- 如果 Jakarta RESTful Web Services root 资源未显式定义范围，它将绑定到请求范围。
-

如果 Jakarta RESTful Web 服务提供程序或 javax.ws.rs.Application 子类未显式定义范围，它将绑定到应用范围。

警告

由于所有未声明范围的 bean 都由 resteasy-cdi 模块修改，因此这也会影响到会话 Bean。因此，如果在规范禁止这些组件为 @RequestScoped 时自动更改无状态会话 Bean 或单例的范围，则会发生冲突。因此，在使用无状态会话 Bean 或单例时，您需要明确定义范围。以后的版本中可能会删除这个要求。

resteasy-cdi 模块与 JBoss EAP 捆绑在一起。因此，不需要单独下载模块或添加任何额外的配置。有关使用 Jakarta Contexts 和带有 Jakarta RESTful Web 服务资源的依赖注入 Bean 的工作示例，请参阅 JBoss EAP 附带的工具包。

2.12. RESTEASY FILTERS 和 INTERCEPTORS

Jakarta RESTful Web Services 有两个不同的拦截器概念：过滤器和拦截器。过滤器主要用于修改或处理传入和传出请求标头或响应标头。它们在请求和响应处理之前和之后执行。

2.12.1. 服务器端过滤器

在服务器端，您有两种不同类型的过滤器：Container RequestFilters 和 ContainerResponseFilters。ContainerRequestFilters 在调用 Jakarta RESTful Web Services 资源方法之前运行。ContainerResponseFilters 在调用 Jakarta RESTful Web Services 资源方法后运行。

此外，有两种类型的 ContainerRequestFilters ：预匹配和后匹配。预匹配 ContainerRequestFilters 由 @PreMatching 注释指定，并在 Jakarta RESTful Web Services 资源方法与传入 HTTP 请求匹配之前执行。匹配后 ContainerRequestFilters 由 @PostMatching 注释指定，并在 Jakarta RESTful Web Services 资源方法与传入 HTTP 请求匹配后执行。

预匹配过滤器通常用于修改请求属性，以更改它与特定资源方法的匹配方式，如 strip .xml 并添加 Accept 标头。ContainerRequestFilters 可以通过调用 ContainerRequestContext.abortWith(Response) 来中止请求。例如，如果过滤器实施了自定义身份验证协议，它可能希望中止。

执行资源类方法后，Jakarta RESTful Web Services 运行所有 ContainerResponseFilters。通过这

些过滤器，您可以在清理并发送到客户端之前修改传出响应。

示例：请求过滤器

```
public class RoleBasedSecurityFilter implements ContainerRequestFilter {
 protected String[] rolesAllowed;
 protected boolean denyAll;
 protected boolean permitAll;

 public RoleBasedSecurityFilter(String[] rolesAllowed, boolean denyAll, boolean permitAll) {
  this.rolesAllowed = rolesAllowed;
  this.denyAll = denyAll;
  this.permitAll = permitAll;
 }

 @Override
 public void filter(ContainerRequestContext requestContext) throws IOException  {
  if (denyAll) {
    requestContext.abortWith(Response.status(403).entity("Access forbidden: role not
allowed").build());
    return;
  }
  if (permitAll) return;
  if (rolesAllowed != null) {
    SecurityContext context =
ResteasyProviderFactory.getContextData(SecurityContext.class);
    if (context != null) {
      for (String role : rolesAllowed) {
        if (context.isUserInRole(role)) return;
      }
      requestContext.abortWith(Response.status(403).entity("Access forbidden: role not
allowed").build());
      return;
    }
  }
  return;
 }
}
```

示例：Response Filter

```
public class CacheControlFilter implements ContainerResponseFilter {
  private int maxAge;

  public CacheControlFilter(int maxAge) {
```

```
      this.maxAge = maxAge;
   }

   public void filter(ContainerRequestContext req, ContainerResponseContext res)
         throws IOException {
      if (req.getMethod().equals("GET")) {
         CacheControl cc = new CacheControl();
         cc.setMaxAge(this.maxAge);
         res.getHeaders().add("Cache-Control", cc);
      }
   }
}
```

2.12.2. 客户端过滤器

有关客户端过滤器的更多信息，请参阅本指南的 Jakarta RESTful Web Services Client API 部分。

2.12.3. RESTEasy Interceptors

2.12.3.1. 拦截 Jakarta RESTful Web Services Invocations

RESTEasy 可以拦截 Jakarta RESTful Web Services 调用，并通过类似于侦听器的对象进行路由。

虽然过滤器修改请求或响应标头，而拦截器会处理消息正文。拦截器在与其对应的读取器或写入器相同的调用堆栈中执行。ReaderInterceptors 围绕执行 MessageBodyReaders 打包.WriterInterceptors 围绕执行 MessageBodyWriters 打包。它们可用于实施特定的内容编码。它们可用于生成数字签名，或者在托管之前或之后发布或预处理 Java 对象模型。

ReaderInterceptors 和 WriterInterceptors 可用于服务器或客户端。它们标有 @Provider，以及 @ServerInterceptor 或 @ClientInterceptor，以便 RESTEasy 知道是否将它们添加到拦截器列表中。

这些拦截器围绕调用 MessageBodyReader.readFrom（）或 MessageBodyWriter.writeTo（）进行 打包。它们可用于嵌套 输出 或 输入 流。

示例：Interceptor

```
@Provider
public class BookReaderInterceptor implements ReaderInterceptor {
```

```
    @Inject private Logger log;
    @Override
    @ReaderInterceptorBinding
    public Object aroundReadFrom(ReaderInterceptorContext context) throws IOException,
WebApplicationException {
        log.info("*** Intercepting call in BookReaderInterceptor.aroundReadFrom()");
        VisitList.add(this);
        Object result = context.proceed();
        log.info("*** Back from intercepting call in BookReaderInterceptor.aroundReadFrom()");
return result;
    }
}
```

拦截器和 MessageBodyReader 或 Writer 在一个大型 Java 调用堆栈中调用。ReaderInterceptorContext.proceed（） 或 WriterInterceptorContext.proceed（） 被调用来进入下一个拦截器；如果没有要调用的拦截器，则调用拦截 器( ) 或 writeTo（） 方法 。此打包允许在对象到达 Reader 或 Writer 之前修改对象，然后在 continue （） 返回后进行清理。

以下示例是服务器端拦截器，它为响应添加一个标头值。

```
@Provider
public class BookWriterInterceptor implements WriterInterceptor {
   @Inject private Logger log;

   @Override
   @WriterInterceptorBinding
   public void aroundWriteTo(WriterInterceptorContext context) throws IOException,
WebApplicationException {
      log.info("*** Intercepting call in BookWriterInterceptor.aroundWriteTo()");
      VisitList.add(this);
      context.proceed();
      log.info("*** Back from intercepting call in BookWriterInterceptor.aroundWriteTo()");
   }
}
```

2.12.3.2. 注册 Interceptor

要在应用中注册 RESTEasy Jakarta RESTful Web Services 拦截器，请将它列在 context-param 元素的 resteasy.providers 参数下的 web.xml 文件中，或者以类或对象形式在 Application. getClasses（）或 Application.get Singletons（） 方法中将它列出。

```
<context-param>
    <param-name>resteasy.providers</param-name>
    <param-value>my.app.CustomInterceptor</paramvalue>
</context-param>
```

```
package org.jboss.resteasy.example;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApp extends Application {

  public java.util.Set<java.lang.Class<?>> getClasses() {
    Set<Class<?>> resources = new HashSet<Class<?>>();
    resources.add(MyResource.class);
    resources.add(MyProvider.class);
    return resources;
  }
}
```

```
package org.jboss.resteasy.example;

import javax.ws.rs.core.Application;
import java.util.HashSet;
import java.util.Set;

public class MyApp extends Application {

    protected Set<Object> singletons = new HashSet<Object>();

    public MyApp() {
        singletons.add(new MyResource());
        singletons.add(new MyProvider());
    }

    @Override
    public Set<Object> getSingletons() {
        return singletons;
    }
}
```

2.12.4. GZIP 压缩与解压缩

RESTEasy 支持 GZIP 压缩和解压缩。为了支持 GZIP 解压缩，客户端框架或 Jakarta RESTful Web Services 服务会自动解压缩具有 gzip Content-Encoding 的消息正文，并且它可以自动将 Accept-Encoding 标头设置为 gzip，因此您不必手动设置此标头。若要支持 GZIP 压缩，如果客户端框架正在发送请求，或者服务器正在发送一个响应（ Content-Encoding 标头设为 gzip ）， RESTEasy 会压缩传出消息。您可以使用 @org.jboss.resteasy.annotation.GZIP 注释来设置 Content-Encoding 标头。

以下示例标记传出消息正文 进行 gzip 压缩的顺序。

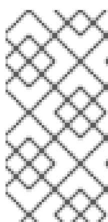
示例：GZIP 压缩

```
@Path("/")
public interface MyProxy {

  @Consumes("application/xml")
  @PUT
  public void put(@GZIP Order order);
}
```

示例：GZIP 压缩服务器响应标签

```
@Path("/")
public class MyService {

  @GET
  @Produces("application/xml")
  @GZIP
  public String getData() {...}
}
```

2.12.4.1. 配置 GZIP 压缩和解压缩



注意

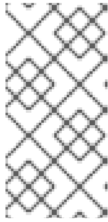
RESTEasy 默认禁用 GZIP 压缩和解压缩，以防止对可能较大但已被攻击者压缩并发送到服务器的实体解压缩。

有三个与 GZIP 压缩和解压缩相关的拦截器：

- org.jboss.resteasy.plugins.interceptors.GZIPDecodingInterceptor ：如果 Content-Encoding 标头存在并且值为 gzip，GZIPDecodingInterceptor 将安装 解压缩邮件正文的输入流。

- org.jboss.resteasy.plugins.interceptors.GZIPEncodingInterceptor ：如果 Content-Encoding 标头存在并且值为 gzip，GZIPEncodingInterceptor 会安装 压缩邮件正文的输出流。

- org.jboss.resteasy.plugins.interceptors.AcceptEncodingGZIPFilter ：如果 Accept-Encoding 标头不存在，Accept EncodingGZIPFilter 会添加值为 gzip 的 Accept-Encoding 标头。如果 Accept-Encoding 标头存在但不包含 gzip，AcceptEncodingGZIPFilter 拦截器将附加该值 gzip。



注意

启用 GZIP 压缩或解压缩并不依赖于 AcceptEncodingGZIPFilter 拦截器的存在性。

启用 GZIP 解压缩器可以为从压缩消息正文中提取的 GZIPDecodingInterceptor 的字节数设置上限。默认限值为 10,000,000。

2.12.4.2. 服务器端 GZIP 配置

您可以通过在类路径上的 javax.ws.rs.ext.Providers 文件中包含它们的类名称来启用拦截器。已定义文件的上限使用 Web 应用上下文参数 resteasy.gzip.max.input 进行设置。如果在服务器端超过这个限制，GZIPDecodingInterceptor 将返回一个带有状态为 413 的响应 - Request Entity Too Large 以及指定上限的消息。

2.12.4.2.1. 客户端 GZIP 配置

您可以通过将 GZIP 拦截器注册到 客户端 或 WebTarget 来启用 GZIP 拦截器。例如：

```
Client client = new ResteasyClientBuilder() // Activate gzip compression on client:
    .register(AcceptEncodingGZIPFilter.class)
    .register(GZIPDecodingInterceptor.class)
    .register(GZIPEncodingInterceptor.class)
    .build();
```

您可以通过创建具有特定值的 GZIPDecodingInterceptor 实例来配置定义文件的上限：

```
Client client = new ResteasyClientBuilder() // Activate gzip compression on client:
    .register(AcceptEncodingGZIPFilter.class)
    .register(new GZIPDecodingInterceptor(256))
    .register(GZIPEncodingInterceptor.class)
    .build();
```

如果在客户端上超过了上限，GZIPDecodingInterceptor 将 引发 ProcessingException，并显示指定上限的消息。

2.12.5. 按资源方法过滤器和 Interceptors

有时，您希望过滤器或拦截器仅针对特定资源方法运行。您可以通过两种不同的方式进行此操作：

- 实施 DynamicFeature 接口。
- 使用 @NameBinding 注释。

实施 DynamicFeature Interface

DynamicFeature 界面包含回调方法，配置(ResourceInfo resourceInfo, FeatureContext context)，后者会针对部署的 Jakarta RESTful Web Services 方法进行调用。ResourceInfo 参数包含有关当前部署的 Jakarta RESTful Web Services 方法的信息。FeatureContext 是 可配置 接口的扩展。您可以使用此参数的 register（） 方法绑定您要分配给此方法的过滤器和拦截器。

示例：使用动态功能接口

```
@Provider
public class AnimalTypeFeature implements DynamicFeature {
   @Override
   public void configure(ResourceInfo info, FeatureContext context) {
      if (info.getResourceMethod().getAnnotation(GET.class) != null)
         AnimalFilter filter = new AnimalFilter();
         context.register(filter);
      }
   }
}
```

在上例中，您使用 AnimalTypeFeature 注册的提供程序必须实施其中一个接口。本例注册了必须实现以下接口之一的供应商 AnimalFilter ： ContainerRequestFilter、ContainerInterceptor、Reader Interceptor、WriterInterceptor 或 Feature。在这种情况下， AnimalFilter 将应用到所有标有 GET 注解的资源方法。详情请参阅 动态功能文档。

使用 @NameBinding 注解

@NameBinding 的工作方式与 Jakarta Contexts 和 Dependency Injection 拦截器非常相似。您使用 @NameBinding 标注自定义注释，然后将该自定义注释应用到过滤器和资源方法。

示例：使用 @NameBinding

```
@NameBinding
public @interface DoIt {}

@DoIt
public class MyFilter implements ContainerRequestFilter {...}

@Path("/root")
public class MyResource {

  @GET
  @DoIt
  public String get() {...}
}
```

详情请参阅 NameBinding 文档。

2.12.6. 排序

通过对过滤器或拦截器类使用 @Priority 注释来完成排序。

2.12.7. 使用过滤器和拦截器处理异常

与过滤器或拦截器关联的例外可以在客户端或服务器端发生。在客户端，您必须处理两种类型的异常： javax.ws.rs.client.ProcessingException 和 javax.ws.rs.client.ResponseProcessingException。如果向服务器发送请求之前 出现了错误，则将在客户端引发 javax.ws.rs.client.ProcessingException。如果处理服务器收到的响应时存在错误，则客户端将 引发 javax.ws.rs.client.ResponseProcessingException。

在服务器端，由过滤器或拦截器抛出的异常处理方式与来自 Jakarta RESTful Web Services 方法的其他 异常处理方式相同，该方法尝试查找引发异常 的异常。有关如何在 Jakarta RESTful Web Services 方法中处理异常的更多详细信息，请参见 例外处理 部分。

2.13. 日志记录 RESTEASY PROVIDERS 和 INTERCEPTORS

RESTEasy 记录 DEBUG 日志记录级别中使用的提供程序和拦截器。您可以使用以下管理 CLI 命令启用与 RESTEasy 相关的所有日志级别：

```
/subsystem=logging/console-handler=CONSOLE:write-attribute(name=level,value=ALL)

/subsystem=logging/logger=org.jboss.resteasy:add(level=ALL)

/subsystem=logging/logger=javax.xml.bind:add(level=ALL)

/subsystem=logging/logger=com.fasterxml.jackson:add(level=ALL)
```

2.14. 异常处理

2.14.1. 创建例外映射器

异常映射器是应用提供的自定义组件，可捕获抛出异常并编写特定的 HTTP 响应。

在创建异常映射器时，您将创建一个标有 @Provider 注释并实施 ExceptionMapper 接口的类。

下面是一个异常映射程序示例：

```
@Provider
public class EJBExceptionMapper implements ExceptionMapper<javax.ejb.EJBException> {
  public Response toResponse(EJBException exception) {
    return Response.status(500).build();
  }
}
```

若要注册异常映射器，可在 resteasy .providers context-param 下将其列出在 web.xml 文件中，或者以编程方式通过 ResteasyProviderFactory 类注册。

2.14.2. 管理内部浏览例外

表 2.2. 例外列表

例外	HTTP 代码	描述
BadRequestException	400	错误请求.请求的格式不正确，或者处理请求输入时出现问题。
UnauthorizedException	401	未授权.如果您使用 RESTEasy 基于角色的注释安全性，则引发安全性异常。
InternalServerErrorException	500	内部服务器错误.

例外	HTTP 代码	描述
MethodNotAllowedException	405	资源没有处理调用的 HTTP 操作的 Jakarta RESTful Web Services 方法。
NotAcceptableException	406	没有 Jakarta RESTful Web Services 方法可以生成 Accept 标头中列出的媒体类型。
NotFoundException	404	没有 Jakarta RESTful Web Services 方法提供请求路径/资源。
ReaderException	400	从 MessageBodyReaders 引发的所有异常均在此例外内封装。如果没有适用于封装异常的 ExceptionMapper，或者例外不是 WebApplicationException，则默认情况下，RESTEasy 会返回 400 代码。
WriterException	500	MessageBodyWriters 引发的所有例外均在此例外内包装。如果没有适用于封装异常的 ExceptionMapper，或者例外不是 WebApplicationException，则默认情况下，RESTEasy 会返回 400 代码。
JAXBUnmarshalException	400	Jakarta XML Binding 提供程序（XML 和 Jackson）抛出这个异常，它可能打包了 JAXBExceptions。此课程扩展了 ReaderException。
JAXBMarshalException	500	Jakarta XML Binding 提供程序（XML 和 Jackson）在写入时抛出此异常，这可能会换掉 JAXBExceptions。此类扩展了 WriterException。
ApplicationException	N/A	将所有异常从应用程序代码中抛出，其运行方式与 InvocationTargetException 相同。如果有一个用于封装异常的例外，则 将用于处理请求。
失败	N/A	内部 RESTEasy 错误.未记录。

例外	HTTP 代码	描述
LoggableFailure	N/A	内部 RESTEasy 错误.已记录.
DefaultOptionsMethodException	N/A	如果用户调用 **HTTP OPTIONS** 并且没有 Jakarta RESTful Web Services 方法，RESTEasy 会抛出此异常来提供默认行为。
UnrecognizedPropertyExceptionHandler	400	当 JSON 数据确定无效时，RESTEasy Jackson 提供程序抛出这一异常。

2.15. 保护 JAKARTA RESTFUL WEB 服务 WEB 服务

RESTEasy 支持 Jakarta RESTful Web 服务方法上的 @RolesAllowed、@PermitAll 和 @DenyAll 注释。但是，您必须启用基于角色的安全性，才能识别这些注释。

2.15.1. 启用基于角色的安全性

按照以下步骤配置 web.xml 文件，以启用基于角色的安全性。

警告

如果应用使用 Jakarta Enterprise Beans，则不要激活基于角色的安全性。Jakarta Enterprise Beans 容器将提供功能，而非 RESTEasy。

为 RESTEasy Jakarta RESTful Web 服务启用基于角色的安全性

1. 在文本编辑器中打开应用的 web.xml 文件。

2. 在 < web-app> 标签内将以下 <context-param > 添加到该文件中。

```
<context-param>
  <param-name>resteasy.role.based.security</param-name>
  <param-value>true</param-value>
</context-param>
```

3. 使用 <security-role> 标签声明 RESTEasy Jakarta RESTful Web Services WAR 文件中使用的所有角色。

```
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
<security-role>
  <role-name>ROLE_NAME</role-name>
</security-role>
```

4. 授权对所有角色的 Jakarta RESTful Web Services 运行时处理的所有 URL 进行访问。

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Resteasy</web-resource-name>
    <url-pattern>/PATH</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ROLE_NAME</role-name>
    <role-name>ROLE_NAME</role-name>
  </auth-constraint>
</security-constraint>
```

5. 为此应用定义适当的登录配置。

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>jaxrs</realm-name>
</login-config>
```

基于角色的安全性已在应用内启用，包含一组定义的角色。

示例：基于角色的安全配置

```
<web-app>

  <context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
  </context-param>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Resteasy</web-resource-name>
```

```
    <url-pattern>/security</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>jaxrs</realm-name>
</login-config>

<security-role>
  <role-name>admin</role-name>
</security-role>
<security-role>
  <role-name>user</role-name>
</security-role>

</web-app>
```

2.15.2. 使用注释保护 Jakarta RESTful Web 服务

若要使用注释保护 Jakarta RESTful Web 服务，请完成以下步骤：

1. 启用基于角色的安全性.

2. 向 Jakarta RESTful Web Services Web 服务添加安全注释。RESTEasy 支持以下注解：

@RolesAllowed

定义哪些角色可以访问该方法。所有角色都应在 web.xml 文件中定义。

@PermitAll

允许 web.xml 文件中定义的所有角色访问该方法。

@DenyAll

拒绝对该方法的所有访问。

以下是使用 @RolesAllowed 注释来指定 admin 角色可以访问 Web 服务的示例：

```
@RolesAllowed("admin")
@Path("/test")
public class TestService {
  ...
}
```

2.15.3. 设置编程安全性

Jakarta RESTful Web 服务包含一个用于收集有关安全请求的安全信息的编程 API。javax.ws.rs.core.SecurityContext 界面有一个方法来确定进行安全 HTTP 调用的用户的身份。它还允许您检查当前用户是否属于某个角色：

```
public interface SecurityContext {

   public Principal getUserPrincipal();
   public boolean isUserInRole(String role);
   public boolean isSecure();
   public String getAuthenticationScheme();
}
```

您可以通过利用 @ Context 注释将它注入字段、setter 方法或资源方法参数来访问 SecurityContext 实例。

```
@Path("test")
public class SecurityContextResource {
    @Context
    SecurityContext securityContext;

    @GET
    @Produces("text/plain")
    public String get() {
        if (!securityContext.isUserInRole("admin")) {
            throw new
WebApplicationException(Response.serverError().status(HttpResponseCodes.SC_UNAUTHO
RIZED)
                    .entity("User " + securityContext.getUserPrincipal().getName() + " is not
authorized").build());
        }
        return "Good user " + securityContext.getUserPrincipal().getName();
    }
}
```

2.16. 异步作业服务 RESTEASY

RESTEasy 异步作业服务旨在为 HTTP 协议添加异步行为。HTTP 是同步协议，但知道异步调用。HTTP 1.1 响应代码 202 Accepted 表示服务器已经接收并接受响应来处理，但处理尚未完成。围绕此构建异步作业服务。

2.16.1. 启用异步作业服务

在 web.xml 文件中启用异步作业服务：

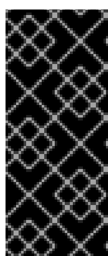
```
<context-param>
    <param-name>resteasy.async.job.service.enabled</param-name>
    <param-value>true</param-value>
</context-param>
```

2.16.2. 配置异步作业

本节涵盖使用 RESTEasy 的异步作业的查询参数示例。

警告

基于角色的安全性无法在异步作业服务中使用，因为它无法移植实施。如果使用异步作业服务，则应用程序安全性必须通过 web.xml 文件中的 XML 声明来实现。



重要

虽然 GET、DELETE 和 PUT 方法可以异步调用，但这破坏了这些方法的 HTTP 1.1 合同。虽然这些调用如果多次调用，可能无法更改资源的状态，但它们会在每次调用时将服务器的状态更改为新作业条目。

asynch 查询参数用于在后台运行调用。将返回 202 接受 的响应，以及位置标头，其 URL 指向后台方法的响应位置。

```
POST http://example.com/myservice?asynch=true
```

上面的示例返回了 202 接受 的响应。它还返回一个位置标头，其 URL 指向后台方法的响应位置。位置标头的示例如下所示：

```
HTTP/1.1 202 Accepted
Location: http://example.com/asynch/jobs/3332334
```

URI 采用以下形式：

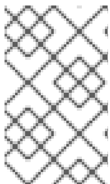
```
/asynch/jobs/{job-id}?wait={milliseconds}|nowait=true
```

GET、POST 和 DELETE 操作可以在此 URL 上执行。

- GET 返回作为响应（如果作业完成）调用的 Jakarta RESTful Web Services 资源方法。如果作业还没有完成，GET 将返回 202 Accepted 响应代码。调用 GET 不会删除作业，因此可以多次调用。

- POST 对作业响应进行读取，并在作业完成后删除作业。

- 调用 DELETE 来手动清理作业队列。



注意

当作业队列已满时，它会自动从内存中驱除最早的作业，而无需调用 DELETE。

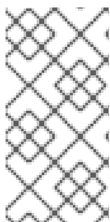
GET 和 POST 操作允许使用 wait 和 nowait 查询参数来定义最长等待时间。如果未指定 wait 参数，则操作将默认为 nowait=true，如果作业未完成，则根本不等待。wait 参数以毫秒为单位定义。

```
POST http://example.com/asynch/jobs/122?wait=3000
```

RESTEasy 支持使用 单 向查询参数来触发和忘记作业。

```
POST http://example.com/myservice?oneway=true
```

上面的示例返回 202 Accepted 响应，但没有创建作业。



注意

异步作业服务的配置参数可在附录的 RESTEasy Asynchronous Job Service Configuration Parameters 部分中找到。

2.17. RESTEASY JAVASCRIPT API

2.17.1. 关于 RESTEasy JavaScript API

RESTEasy 可以生成 JavaScript API，它使用 AJAX 调用来调用 Jakarta RESTful Web 服务操作。每个 Jakarta RESTful Web Services 资源类都将生成名称与声明类或接口相同的 JavaScript 对象。JavaScript 对象包含每个 Jakarta RESTful Web Services 方法作为属性。

```
@Path("foo")
public class Foo {

  @Path("{id}")
  @GET
  public String get(@QueryParam("order") String order, @HeaderParam("X-Foo") String
header,
    @MatrixParam("colour") String colour, @CookieParam("Foo-Cookie") String cookie) {
  }

  @POST
  public void post(String text) {
  }
}
```

以下 JavaScript 代码使用上例中生成的 Jakarta RESTful Web 服务 API：

```
var text = Foo.get({order: 'desc', 'X-Foo': 'hello', colour: 'blue', 'Foo-Cookie': 123987235444});
Foo.post({$entity: text});
```

每一 JavaScript API 方法将可选对象用作单一参数，其中每个属性都是一个 Cookie、标头、路径、查询或表单参数（根据名称或 API 参数属性标识）。有关 API 参数属性的详情，请参阅 RESTEasy Javascript API 参数 附录。

2.17.1.1. 启用 RESTEasy JavaScript API Servlet

RESTEasy JavaScript API 默认禁用。按照以下步骤，通过更新 web.xml 文件来启用它。

1. 在文本编辑器中打开应用的 web.xml 文件。

2. 在 web-app 标签中添加以下配置：

```
<servlet>
    <servlet-name>RESTEasy JSAPI</servlet-name>
    <servlet-class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>RESTEasy JSAPI</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

2.17.1.2. 构建 AJAX 查询

RESTEasy JavaScript API 可用于手动构建请求。以下是此行为的一些示例：

示例：用于覆盖 RESTEasy JavaScript API 客户端的 REST 对象

```
// Change the base URL used by the API:
REST.apiURL = "http://api.service.com";

// log everything in a div element
REST.log = function(text) {
 jQuery("#log-div").append(text);
};
```

REST 对象包含以下读写属性：

- apiURL ：默认设置为 Jakarta RESTful Web Services root URL。在构建请求时供每个 JavaScript 客户端 API 功能使用。
- log ：设置为 功能（字符串），以便接收 RESTEasy 客户端 API 日志。如果要调试客户端 API 并将日志放在可以查看它们的位置，这很有用。

示例：使用 REST.Request（）方法构建自定义请求的类

```
var r = new REST.Request();
r.setURI("http://api.service.com/orders/23/json");
r.setMethod("PUT");
r.setContentType("application/json");
r.setEntity({id: "23"});
r.addMatrixParameter("JSESSIONID", "12309812378123");
r.execute(function(status, request, entity) {
  log("Response is " + status);
});
```

2.18. 用于修改资源元数据的 RESTEASY SPI

JBoss EAP 提供 RESTEasy 服务提供商接口(SPI)，用于修改使用 ResourceBuilder 创建的资源类元数据。在处理 Jakarta RESTful Web 服务部署时，RESTEasy 使用 ResourceBuilder 为 Jakarta RESTful Web 服务资源创建元数据。此类元数据使用软件包 org.jboss.resteasy.spi.metadata 中的 元数据 SPI 定义，特别是 ResourceClass 接口：

```
package org.jboss.resteasy.spi.metadata;

public interface ResourceClass
{
  String getPath();

  Class<?> getClazz();

  ResourceConstructor getConstructor();

  FieldParameter[] getFields();

  SetterParameter[] getSetters();

  ResourceMethod[] getResourceMethods();

  ResourceLocator[] getResourceLocators();
}
```

RESTEasy 允许通过提供 ResourceClassProcessor 接口的实施来自定义元数据生成。以下示例演示了这个 SPI 的用法：

```
package org.jboss.resteasy.test.core.spi.resource;

import org.jboss.logging.Logger;
import org.jboss.resteasy.spi.metadata.ResourceClass;
import org.jboss.resteasy.spi.metadata.ResourceClassProcessor;

import javax.ws.rs.ext.Provider;
```

```
@Provider
public class ResourceClassProcessorImplementation implements ResourceClassProcessor {

    protected static final Logger logger =
Logger.getLogger(ResourceClassProcessorImplementation.class.getName());
@Override
    public ResourceClass process(ResourceClass clazz) {
        logger.info(String.format("ResourceClassProcessorImplementation#process method
called on class %s",
                        clazz.getClazz().getSimpleName()));
        String clazzName = clazz.getClazz().getSimpleName();
        if (clazzName.startsWith("ResourceClassProcessorEndPoint")
                || clazzName.equals("ResourceClassProcessorProxy")
                || clazzName.equals("ResourceClassProcessorProxyEndPoint")) {
            return new ResourceClassProcessorClass(clazz);
        }
        return clazz;
    }
```

使用 ResteasyProviderFactory 类存储的新处理器作为标有提供程序的常规 Jakarta RESTful Web 服务解决。它们允许使用可用于各种高级场景的自定义版本嵌套资源元数据类，例如：

- 向资源中添加其他资源方法或定位器.
- 修改 HTTP 方法.
- 修改 @Produces 或 @Consumes 媒体类型.

2.19. MICROPROFILE REST 客户端

重要

MicroProfile REST 客户端仅作为技术预览提供。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

JBoss EAP 7.4 支持 MicroProfile REST 客户端 1.4.x，它构建于 Jakarta RESTful Web Services 2.1 客户端 API，以提供通过 HTTP 调用 RESTful 服务的类型安全方法。MicroProfile 类型 Safe REST

客户端定义为 Java 接口。利用 MicroProfile REST 客户端，您可以使用可执行代码编写客户端应用。

MicroProfile REST 客户端启用：

- 直观的语法
- 供应商的程序注册
- 供应商声明注册
- 标头声明规格
- 在服务器中传播标头
- ResponseExceptionMapper
- Jakarta 上下文和依赖注入集成

2.19.1. 直观的语法

MicroProfile REST 客户端启用分布式对象通信版本，它也在 CORBA、Java 远程方法调用(RMI)、JBoss 远程方法传递项目和 RESTEasy 中实施。例如，考虑资源：

```
@Path("resource")
public class TestResource {
  @Path("test")
  @GET
  String test() {
    return "test";
  }
 }
```

访问 TestResource 类的 Jakarta RESTful Web 服务原生方式是：

```
Client client = ClientBuilder.newClient();
String response = client.target("http://localhost:8081/test").request().get(String.class);
```

但是，Microprofile REST 客户端通过直接调用 test（） 方法来支持更直观的语法：

```
@Path("resource")
public interface TestResourceIntf {
    @Path("test")
    @GET
    public String test();
}

TestResourceIntf service = RestClientBuilder.newBuilder()
                              .baseUrl(http://localhost:8081/))
                              .build(TestResourceIntf.class);
String s = service.test();
```

在上例中，让 TestResource 类的调用在使用 TestResource Intf 类时变得更容易，如调用 service.test（） 所示。

以下示例是 TestResourceIntf 类的更详细版本：

```
@Path("resource")
public interface TestResourceIntf2 {
   @Path("test/{path}")mes("text/plain")
   @Produces("text/html")
   @POST
   public String test(@PathParam("path") String path, @QueryParam("query") String query,
String entity);
}
```

调用 service.test("p", "q", "e") 方法会产生一个类似如下的 HTTP 消息：

```
POST /resource/test/p/?query=q HTTP/1.1
Accept: text/html
Content-Type: text/plain
Content-Length: 1

e
```

2.19.2. 供应商的程序注册

借助 MicroProfile REST 客户端，您还可以通过注册提供程序来配置客户端环境。例如：

```
TestResourceIntf service = RestClientBuilder.newBuilder()
                            .baseUrl(http://localhost:8081/))
                            .register(MyClientResponseFilter.class)
                            .register(MyMessageBodyReader.class)
                            .build(TestResourceIntf.class);
```

2.19.3. 供应商的声明注册

您还可以通过在目标接口中添加 org.eclipse.microprofile.rest.client.annotation.RegisterProvider 注解，在目标接口中添加 org.eclipse.microprofile.rest.client.annotation.RegisterProvider 注解，以声明方式注册供应商：

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
@RegisterProvider(MyMessageBodyReader.class)
public interface TestResourceIntf2 {
  @Path("test/{path}")
  @Consumes("text/plain")
  @Produces("text/html")
  @POST
  public String test(@PathParam("path") String path, @QueryParam("query") String query,
String entity);
}
```

使用注释 声明 MyClientResponseFilter 类和 MyMessageBodyReader 类无需调用 RestClientBuilder.register（） 方法。

2.19.4. Headers 声明规格

您可以使用以下方法为 HTTP 请求指定标头：

- 通过标注其中一个资源方法参数：
- 声明使用 org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam 注释。

以下示例演示了设置标头，方法是使用注释 @HeaderValue 为其中一个资源方法参数添加注解：

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
String contentLang(@HeaderParam(HttpHeaders.CONTENT_LANGUAGE) String
contentLanguage, String subject);
```

以下示例演示了使用 org.eclipse.microprofile.rest.client.annotation.ClientHeaderParam 注释设置标头：

```
@POST
@Produces(MediaType.TEXT_PLAIN)
@Consumes(MediaType.TEXT_PLAIN)
@ClientHeaderParam(name=HttpHeaders.CONTENT_LANGUAGE, value="{getLanguage}")
String contentLang(String subject);

default String getLanguage() {
  return ...;
}
```

2.19.5. 在服务器上传播标头

org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory 的实例如果激活，可以批量传输传入的标头到传出请求。默认实例 org.eclipse.microprofile.rest.client.ext.DefaultClientHeadersFactoryImpl 将返回一个映射，这些映射由逗号分隔的配置属性 org.eclipse.microprofile.rest.client.propagateHeaders 中列出的。以下是实例化 ClientHeadersFactory 接口的规则：

- 在 Jakarta RESTful Web 服务请求中调用的 ClientHeadersFactory 实例必须支持注入标有 @Context 的字段和方法。
- 由 Jakarta Contexts 和 Dependency Injection 管理的 ClientHeadersFactory 实例必须使用适当的 Jakarta Contexts 和 Dependency Injection 管理的实例。它还必须支持 @Inject 注入。

org.eclipse.microprofile.rest.client.ext.ClientHeadersFactory 接口定义如下：

```
public interface ClientHeadersFactory {

/**
 * Updates the HTTP headers to send to the remote service. Note that providers
 * on the outbound processing chain could further update the headers.
 *
 * @param incomingHeaders - the map of headers from the inbound Jakarta RESTful Web
Services request. This will
 * be an empty map if the associated client interface is not part of a Jakarta RESTful Web
Services request.
 * @param clientOutgoingHeaders - the read-only map of header parameters specified on the
 * client interface.
 * @return a map of HTTP headers to merge with the clientOutgoingHeaders to be sent to
 * the remote service.
 */
```

```
MultivaluedMap<String, String> update(MultivaluedMap<String, String> incomingHeaders,
                                      MultivaluedMap<String, String> clientOutgoingHeaders);
}
```

有关 ClientHeadersFactory 接口的更多信息，请参阅 ClientHeadersFactory Javadoc。

2.19.6. ResponseExceptionMapper

org.eclipse.microprofile.rest.client.ext.ResponseExceptionMapper 类是 Jakarta RESTful Web Services 中定义的 javax.ws.rs.ext.ExceptionMapper 类的客户端反转。也就是说，Exception Mapper.toResponse（） 方法将服务器端处理期间引发的 Exception 类转换为 Response 类，即 ResponseExceptionMapper.toThrowable（） 方法将客户端上收到的 Response 类转换为 Exception 类。

您可以以编程或声明方式注册 ResponseExceptionMapper 类。如果没有注册的 ResponseExceptionMapper 类，默认的 ResponseExceptionMapper 类会将任何响应 映射到 一个 WebApplicationException 类。

2.19.7. Jakarta 上下文和依赖注入集成

在 MicroProfile REST 客户端中，您必须使用 @RegisterRestClient 类为作为 Jakarta 上下文和依赖注入 Bjection Bjection Bjection Bjection Bjection Ban 标注任何接口。例如：

```
@Path("resource")
@RegisterProvider(MyClientResponseFilter.class)
public static class TestResourceImpl {
    @Inject TestDataBase db;

    @Path("test/{path}")
    @Consumes("text/plain")
    @Produces("text/html")
    @POST
    public String test(@PathParam("path") String path, @QueryParam("query")
    String query, String entity) {
      return db.getByName(query);
    }
  }
  @Path("database")
  @RegisterRestClient
  public interface TestDataBase {

    @Path("")
    @POST
    public String getByName(String name);
  }
```

此处，MicroProfile REST 客户端实施为 TestDataBase 类服务创建一个客户端，让 TestResourceImpl 类能够轻松访问。但是，它不包括有关 TestDataBase 类实施路径的信息。此信息可以由可选 @Reg- isterProvider 参数 baseUri 提供：

```
@Path("database")
@RegisterRestClient(baseUri="https://localhost:8080/webapp")
public interface TestDataBase {
  @Path("")
  @POST
  public String getByName(String name);
}
```

这表示您可以访问 TestDataBase 的实现，地址 为 https://localhost:8080/webapp。您还可以向外部提供以下系统变量信息：

```
<fully qualified name of TestDataBase>/mp-rest/url=<URL>
```

例如，以下命令表示您可以访问位于 https://localhost:8080/webapp 的 com.bluemondiamond.TestDatabase 类的实施：

```
com.bluemonkeydiamond.TestDatabase/mp-rest/url=https://localhost:8080/webapp
```

2.20. 对 COMPLETIONSTAGE 类型的支持

Jakarta RESTful Web Services 2.1 规范支持通过返回 CompletionStage 而非 @Suspended 注释来声明异步资源方法。

每当资源方法返回它订阅的 CompletionStage 时，请求都会暂停。只有在 CompletionStage 类型为 CompletionStage 类型时才会恢复请求：

- 解析为值，然后被视为方法的返回值。
- 异常被视为错误情况，处理异常的方式就像资源方法抛出一样。

以下是使用 CompletionStage 的异步处理示例：

```
public class SimpleResource
{
```

```
    @GET
    @Path("basic")
    @Produces("text/plain")
    public CompletionStage<Response> getBasic() throws Exception
    {
      final CompletableFuture<Response> response = new CompletableFuture<>();
      Thread t = new Thread()
      {
        @Override
        public void run()
        {
          try
          {
            Response jaxrs = Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
            response.complete(jaxrs);
          }
          catch (Exception e)
          {
            response.completeExceptionally(e);
          }
        }
      };
      t.start();
      return response;
    }
  }
```

2.21. 扩展 RESTEASY 支持异步请求处理和 REACTIVE RETURN 类型

重要

扩展 RESTEasy 支持仅作为技术预览提供。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

如需有关 技术预览功能支持范围 的信息，请参阅红帽客户门户网站中的技术预览功能支持范围。

2.21.1. 可插拔响应类型

Jakarta RESTful Web Services 2.1 可扩展，可支持各种被动库。RESTEasy 的可选模块 resteasy-rxjava2 支持以下被动类型：

- io.reactivex.Single ：与 CompletionStage 类似，因为它最多有一个潜在值。

- io.reactivex.Flowable ：Implements io.reactivex.Publisher.

- io.reactivex.Observable ：与 Flowable 类似，只是不支持反压缩，订阅者可以通过调用 Subscription.request（） 来控制从生产者收到的负载。

如果您导入 resteasy-rxjava2，您可以从服务器端的资源方法返回这些被动类型，并在客户端端接收它们。

resteasy-rxjava2 模块支持以下三个类分别访问客户端一侧的 Singles、Oservables 和 Flowables：

- org.jboss.resteasy.rxjava2.SingleRxInvoker

- org.jboss.resteasy.rxjava2.FlowableRxInvoker

- org.jboss.resteasy.rxjava2.ObservableRxInvoker

2.21.2. 其他活跃类的扩展

RESTEasy 实施框架，支持其他被动类的扩展。在服务器端，当资源方法返回 CompletionStage 类型时，RESTEasy 会使用 org.jboss.resteasy.core.AsyncResponseConsumer 类订阅它。当 CompletionStage 完成后，它会调用 CompletionStageResponseConsumer.accept（），它会将结果发回到客户端。

RESTEasy 内置了对 CompletionStage 的支持。我们可以通过提供一种将单机转换为完成阶段的机制，将支持扩展到 单 类 。在 resteasy-rxjava2 模块中，org.jboss.resteasy.rxjava2.SingleProvider 实施 org.jboss.resteasy.spi.AsyncResponseProvider<Single<?>> 接口提供此机制：

```
public interface AsyncResponseProvider<T> {
  public CompletionStage toCompletionStage(T asyncResponse);
}
```

假定为 SingleProvider 类，RESTEasy 可以单选一个 单 级，将它转换为 CompletionStage ResponseConsumer，然后使用 CompletionStageResponseConsumer 来处理 单 台最终值。类似地，当资源方法返回流的被动类（如 Flowable ）时，RESTEasy 订阅它，接收数据元素流并将其发送到客户端。AsyncResponseConsumer 具有多个支持类，各自实施不同的流模式。

例如，AsyncResponseConsumer.Async GeneralStreamingSseResponseConsumer 处理常规流和 SSE 流。订阅通过调用 org.reactivestreams.Publisher.subscribe（） 完成，因此需要一种机制来将流向 发布 程序，例如：也就是说，调用 org.jboss.resteasy.spi.AsyncStreamProvider<Flowable> 的实现，其定义 AsyncStreamProvider，如下例所示：

```
public interface AsyncStreamProvider<T> {
  public Publisher toAsyncStream(T asyncResponse);
}
```

在 resteasy-rxjava2 模块中，org.jboss.resteasy.FlowableProvider 为 Flowable 提供这种机制。

这意味着，在服务器端，您可以通过为流或 AsyncResponse Provider 接口声明对流或 AsyncResponseProvider 接口的 @ Provider 注释来添加对其他被动类型的支持。这两个接口都有一个方法，可将新的被动类型分别转换为 Publisher 或 CompletionStage （流）或单个值。

在客户端，Jkarta RESTful Web Services 2.1 提出了两个支持被动类的要求：

- 支持 CompletionStage 作为 javax.ws.rs.client.CompletionStageRxInvoker 接口的实施。

- 支持注册实施的供应商的可扩展性：

```
public interface RxInvokerProvider<T extends RxInvoker> {
   public boolean isProviderFor(Class<T> clazz);
   public T getRxInvoker(SyncInvoker syncInvoker, ExecutorService executorService);
}
```

注册 RxInvokerProvider 后，您可以通过调用 javax.ws.rs.client.Invocation.Builder 方法来请求 RxInvoker：

```
public <T extends RxInvoker> T rx(Class<T> clazz);
```

您可以使用 RxInvoker 进行调用，以返回适当的被动类。例如：

```
FlowableRxInvoker invoker =
client.target(generateURL("/get/string")).request().rx(FlowableRxInvoker.class);
Flowable<String> flowable = (Flowable<String>) invoker.get();
```

RESTEasy 为实施 RxInvokers 提供部分支持。例如，上面提到的 SingleProvider 还实施了 org.jboss.resteasy.spi.AsyncClientResponseProvider<Single<?>>，其中 AsyncClientResponseProvider 定义为以下内容：

```
public interface AsyncClientResponseProvider<T> {
  public T fromCompletionStage(CompletionStage<?> completionStage);
}
```

2.21.3. 被动客户端 API

RESTEasy 定义名为 RxInvoker 的新调用者类型，以及此类型的默认实施，名为 CompletionStageRxInvoker。CompletionStageRxInvoker 实施 Java 8 的界面 完成阶段。此接口声明了大量专门用于管理异步计算的方法。

2.21.4. 异步过滤器

如果必须暂停执行过滤器直到有特定资源可用，您可以将其转换为异步过滤器。关闭请求异步不需要对您的资源方法声明或额外的过滤器声明进行任何更改。

要将过滤器的执行异步切换，您必须进行 cast：

- ContainerRequestContext to SuspendableContainerRequestContext 用于 pre 和 post 请求过滤器。
- ContainerResponseContext 转换为用于响应过滤器 的 SuspendableContainerResponseContext。

这些上下文对象可以通过调用 suspend（） 方法将当前过滤器的执行转换为异步。异步后，过滤器链会被暂停，只有在上下文对象中调用以下方法之一后才会恢复：

- abortWith(Response)：终止过滤器链，将给定的 Response 返回给客户端。这只适用于 ContainerRequestFilter。
- restore（） ：通过调用下一个过滤器来恢复过滤器链的执行。
- Restore（可浏览） ：通过抛出给定异常来中止执行过滤器链。这的行为就像过滤器是同步的，并且三个给出的异常。

2.21.5. 代理

代理是 RESTEasy 扩展，支持直观的编程风格，用特定于应用的接口调用来取代通用 Jakarta RESTful Web 服务调用者调用。代理框架扩展为包括 CompletionStage 和 RxJava2 类型 Single、Observable 和 Flowable。以下示例演示了 RESTEasy 代理如何工作：

示例 1：

```
@Path("")
public interface RxCompletionStageResource {

  @GET
  @Path("get/string")
  @Produces(MediaType.TEXT_PLAIN)
  public CompletionStage<String> getString();
}

@Path("")
public class RxCompletionStageResourceImpl {

  @GET
  @Path("get/string")
  @Produces(MediaType.TEXT_PLAIN)
  public CompletionStage<String> getString() { .... }
}

public class RxCompletionStageProxyTest {

  private static ResteasyClient client;
  private static RxCompletionStageResource proxy;

  static {
    client = new ResteasyClientBuilder().build();
    proxy = client.target(generateURL("/")).proxy(RxCompletionStageResource.class);
  }

  @Test
  public void testGet() throws Exception {
    CompletionStage<String> completionStage = proxy.getString();
    Assert.assertEquals("x", completionStage.toCompletableFuture().get());
  }
}
```

示例 2：

```
public interface Rx2FlowableResource {
```

```
    @GET
    @Path("get/string")
    @Produces(MediaType.TEXT_PLAIN)
    @Stream
    public Flowable<String> getFlowable();
}

@Path("")
public class Rx2FlowableResourceImpl {

    @GET
    @Path("get/string")
    @Produces(MediaType.TEXT_PLAIN)
    @Stream
    public Flowable<String> getFlowable() { ... }
}

public class Rx2FlowableProxyTest {

    private static ResteasyClient client;
    private static Rx2FlowableResource proxy;

    static {
        client = new ResteasyClientBuilder().build();
        proxy = client.target(generateURL("/")).proxy(Rx2FlowableResource.class);
    }

    @Test
    public void testGet() throws Exception {
        Flowable<String> flowable = proxy.getFlowable();
        flowable.subscribe(
            (String o) -> stringList.add(o),
            (Throwable t) -> errors.incrementAndGet(),
            () -> latch.countDown());
        boolean waitResult = latch.await(30, TimeUnit.SECONDS);
        Assert.assertTrue("Waiting for event to be delivered has timed out.", waitResult);
        Assert.assertEquals(0, errors.get());
        Assert.assertEquals(xStringList, stringList);
    }
}
```

第 3 章 开发 JAKARTA XML WEB 服务

Jakarta XML Web 服务定义了 WSDL 和 Java 之间的映射，以及用于访问 Web 服务并发布它们的类。JBossWS 实施 Jakarta XML Web 服务 2.3，用户可以参考这些服务用于任何与供应商无关的 Web 服务使用需求。

3.1. 使用 JAKARTA XML WEB 服务工具

以下 Jakarta XML Web Services 命令行工具包含在 JBoss EAP 发行版中：这些工具可用于 服务器和 客户端开发。

表 3.1. Jakarta XML Web Services 命令行工具

命令	描述
wsprovide	生成 Jakarta XML Web 服务便携式工件，并提供抽象合同。用于底层开发.
wsconsume	使用抽象合同（WSDL 和架构文件），并为服务器和客户端生成工件。用于自上而下和客户端开发.

有关使用这些工具的更多详细信息，请参阅 Jakarta XML Web Services 工具。

3.1.1. 服务器侧开发策略

在服务器端开发 Web 服务端点时，您可以选择从 Java 代码（称为 *底部开发）* 或 WSDL 开始，后者定义您的服务，称为 *自上而下开发*。如果这是一种新服务，即没有现有合同，则下向上的方法是最快的路线；您只需向类添加一些注释即可启动和运行服务。但是，如果您在开发服务时已定义了合同，使用自上而下的方法会更加简单，因为该工具可以为您生成注释的代码。

底部向上使用案例：

- 将已存在的 Jakarta Enterprise Beans 3 bean 作为 Web 服务公开。

- 提供新的服务，并且您希望为您生成合同。

上下使用案例：

- 替换现有 Web 服务的实施，您不能破坏与旧客户端的兼容性。
- 公开符合第三方指定合同的服务，例如供应商重新调用您已定义的协议。
- 创建遵循预先开发的 XML 架构和 WSDL 的服务。

使用 wsprovide 的底部策略

底层策略涉及为您的服务开发 Java 代码，然后使用 Jakarta XML Web 服务 注释进行标注。这些注解可用于自定义为您的服务生成的合同。例如，您可以更改操作名称来映射到您喜欢的任何内容。但是，所有注释都具有明智的默认值，因此只需要 @WebService 注释。

这就像创建单个类一样简单：

```
package echo;

@javax.jws.WebService
public class Echo {

   public String echo(String input) {
      return input;
   }
}
```

可以使用此类构建部署，这是在 JBossWS 上部署的唯一 Java 代码。在部署时，会为您生成 WSDL 和所有其他 Java 构件，称为 *wrapper* 类。

wsprovide 工具的主要用途是生成可移植 Jakarta XML Web Services 工件。此外，它也可用于为您的服务提供 WSDL 文件。这可以通过使用 -w 选项调用 wsprovide 来获取：

```
$ javac -d . Echo.java
$ EAP_HOME/bin/wsprovide.sh --classpath=. -w echo.Echo
```

检查 WSDL 会显示一个名为 EchoService 的服务 ：

```
<wsdl:service name="EchoService">
  <wsdl:port name="EchoPort" binding="tns:EchoServiceSoapBinding">
    <soap:address location="http://localhost:9090/EchoPort"/>
  </wsdl:port>
</wsdl:service>
```

如预期的那样，该服务会定义一个操作，echo：

```
<wsdl:portType name="Echo">
  <wsdl:operation name="echo">
    <wsdl:input name="echo" message="tns:echo">
    </wsdl:input>
    <wsdl:output name="echoResponse" message="tns:echoResponse">
    </wsdl:output>
  </wsdl:operation>
</wsdl:portType>
```

在部署时，您不需要运行此工具。您只需要为服务生成可移植工件或抽象合同即可。

可在简单的 web.xml 文件中为部署创建 POJO 端点：

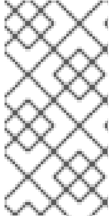
```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>Echo</servlet-name>
    <servlet-class>echo.Echo</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Echo</servlet-name>
    <url-pattern>/Echo</url-pattern>
  </servlet-mapping>
</web-app>
```

web.xml 和单个 Java 类现在可以用于创建 WAR：

```
$ mkdir -p WEB-INF/classes
$ cp -rp echo WEB-INF/classes/
$ cp web.xml WEB-INF
$ jar cvf echo.war WEB-INF
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/echo/Echo.class(in = 340) (out= 247)(deflated 27%)
adding: WEB-INF/web.xml(in = 576) (out= 271)(deflated 52%)
```

然后可以将 WAR 部署到 JBoss EAP。这会在内部调用 wsprovide，这将生成 WSDL。如果部署成

功，并且您正在使用默认设置，它应在管理控制台中可用。

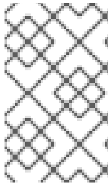


注意

对于可移植的 Jakarta XML Web 服务部署，可以添加之前生成的打包程序类到部署中。

使用 wssume 的自顶策略

自上而下的开发策略从该服务的抽象合同开始，其中包括 WSDL 文件和零个或多个架构文件。然后，使用 wssume 工具来消耗此合同，生成注解的 Java 类，以及定义它的可选来源。



注意

ws Consume 可能会在 Unix 系统中出现符号链接问题。

使用底部示例中的 WSDL 文件，可以生成遵循此服务的新 Java 实施。k 选项 传递到 ws Consume 以保留 生成的 Java 源文件，而不是仅提供 Java 类：

```
$ EAP_HOME/bin/wsconsume.sh -k EchoService.wsdl
```

下表显示了每个生成的文件的用途：

表 3.2. 生成的文件

File	用途
Echo.java	服务端点接口
EchoResponse.java	用于响应消息的 wrapper bean
EchoService.java	仅供 Jakarta XML Web 服务客户端使用
Echo_Type.java	用于请求消息的 wrapper bean
ObjectFactory.java	Jakarta XML Binding XML Registry
package-info.java	Jakarta XML Binding 软件包注解的拥有者

检查服务端点接口发现的注释比底部上例中手动编写的类中更明确，但它们评估到同一合同。

```
@WebService(targetNamespace = "http://echo/", name = "Echo")
@XmlSeeAlso({ObjectFactory.class})
public interface Echo {

   @WebMethod
   @RequestWrapper(localName = "echo", targetNamespace = "http://echo/", className =
"echo.Echo_Type")
   @ResponseWrapper(localName = "echoResponse", targetNamespace = "http://echo/",
className = "echo.EchoResponse")
   @WebResult(name = "return", targetNamespace = "")
   public java.lang.String echo(
      @WebParam(name = "arg0", targetNamespace = "")
      java.lang.String arg0
   );
}
```

除了打包之外缺少的唯一部分是实施类，现在可以使用上述界面编写。

```
package echo;

@javax.jws.WebService(endpointInterface="echo.Echo")
public class EchoImpl implements Echo {
   public String echo(String arg0) {
      return arg0;
   }
}
```

3.1.2. 客户端开发策略

在详细讲解客户端之前，务必要了解 Web 服务的核心分离概念。Web 服务不是最适合内部 RPC 服务，即使这些服务可以以这种方式使用。有更好的技术可以做到这一点，如 CORBA 和 RMI。Web 服务专为可互操作的粗粒度交流而设计。不期望或保证参与 Web 服务交互的各方将位于任何特定的位置，在任何特定的操作系统上运行，或者使用任何特定编程语言编写。因此，明确分隔客户端和服务器实施非常重要。他们唯一应当具有的共性是抽象合同定义。如果出于任何原因，您的软件不遵循此主体，那么您不应使用 Web 服务。由于上述原因，建议使用自上而下的方法开发客户端，即使客户端在同一服务器上运行也是如此。

使用 wssume 的自顶策略

本节重复了服务器端上下一节的流程，但它使用了已部署的 WSDL。这是为 soap :address （如下所示）检索正确的值，该值在部署时计算。如有必要，可在 WSDL 中手动编辑这个值，但您必须小心提供正确的路径。

示例 ：已部署 WSDL 中的 soap:address

```
<wsdl:service name="EchoService">
  <wsdl:port name="EchoPort" binding="tns:EchoServiceSoapBinding">
    <soap:address location="http://localhost.localdomain:8080/echo/Echo"/>
  </wsdl:port>
</wsdl:service>
```

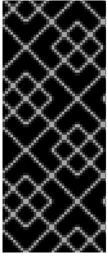
使用 ws Consume 为部署的 WSDL 生成 Java 类。

```
$ EAP_HOME/bin/wsconsume.sh -k http://localhost:8080/echo/Echo?wsdl
```

注意 EchoService.java 类如何存储从中获取 WSDL 的位置。

```
@WebServiceClient(name = "EchoService",
                  wsdlLocation = "http://localhost:8080/echo/Echo?wsdl",
                  targetNamespace = "http://echo/")
public class EchoService extends Service {

    public final static URL WSDL_LOCATION;

    public final static QName SERVICE = new QName("http://echo/", "EchoService");
    public final static QName EchoPort = new QName("http://echo/", "EchoPort");

    ...

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort() {
        return super.getPort(EchoPort, Echo.class);
    }

    @WebEndpoint(name = "EchoPort")
    public Echo getEchoPort(WebServiceFeature... features) {
        return super.getPort(EchoPort, Echo.class, features);
    }
}
```

如您所见，此生成的类扩展了 Jakarta XML Web Services javax.xml.ws.Service 中的主要客户端入口点。虽然您可以直接使用 服务，但这要简单得多，因为它能为您提供配置信息。注意 getEchoPort（） 方法，它会返回我们服务端点接口的实例。然后，可以通过对返回的接口调用方法来调用任何 Web 服务操作。



重要

不要引用生产应用中的远程 WSDL URL。这会导致每次实例化 Service 对象时网络 I/O。相反，请使用已保存的本地副本上的 工具，或者使用构造器的 URL 版本来提供新的 WSDL 位置。

编写并编译客户端：

```
import echo.*;

public class EchoClient {

  public static void main(String args[]) {

    if (args.length != 1) {
       System.err.println("usage: EchoClient <message>");
       System.exit(1);
    }

    EchoService service = new EchoService();
    Echo echo = service.getEchoPort();
    System.out.println("Server said: " + echo.echo(args0));
  }
}
```

您可以通过设置 ENDPOINT_ADDRESS_PROPERTY，在运行时更改操作的端点地址，如下所示：

```
EchoService service = new EchoService();
Echo echo = service.getEchoPort();

/* Set NEW Endpoint Location */
String endpointURL = "http://NEW_ENDPOINT_URL";
BindingProvider bp = (BindingProvider)echo;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
endpointURL);

System.out.println("Server said: " + echo.echo(args0));
```

3.2. JAKARTA XML WEB 服务端点

3.2.1. 关于 Jakarta XML Web 服务端点

Jakarta XML Web Services Web 服务端点是 Web 服务的服务器组件。客户端和其他 Web 服务使用称为简单对象访问协议(SOAP)的 XML 语言通过 HTTP 协议与其通信。端点本身部署到 JBoss EAP 容器中。

可以通过以下两种方式之一创建 WSDL 描述符：

- 手动编写 WSDL 描述符.
- 使用 Jakarta XML Web 服务注释，自动创建 WSDL 描述符。这是创建 WSDL 描述符的最常用方法。

端点实施带有 Jakarta XML Web Services 注释并部署到服务器。服务器会自动生成并发布 WSDL 格式的抽象合同，供客户端使用。所有 marshalling 和 unmarshalling 均委派至 Jakarta XML Binding 服务。

端点本身可能是 Plain Old Java 对象(POJO)或 Jakarta EE Web 应用。您还可以使用 Jakarta Enterprise Beans 3 无状态会话 Bean 公开端点。它打包成 Web 存档(WAR)文件。打包端点的规范在 Jakarta Web Services 元数据规范 2.1 中定义。

示例：POJO 端点

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class JSEBean {
    @WebMethod
    public String echo(String input) {
        ...
    }
}
```

示例：Web 服务端点

```
<web-app ...>
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-class>org.jboss.quickstarts.ws.jaxws.samples.jsr181pojo.JSEBean01</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
```

```
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

以下 Jakarta Enterprise Beans 3 无状态会话 Bean 在远程接口上公开了相同的方法，以及端点操作。

```
@Stateless
@Remote(EJB3RemoteInterface.class)

@WebService

@SOAPBinding(style = SOAPBinding.Style.RPC)
public class EJB3Bean implements EJB3RemoteInterface {
    @WebMethod
    public String echo(String input) {
        ...
    }
}
```

服务端点接口

Jakarta XML Web 服务通常实施 Java 服务端点接口(SEI)，它可以从 WSDL 端口类型（直接或使用注释）映射。此 SEI 提供了一个高级抽象，隐藏了 Java 对象及其 XML 表示之间的详细信息。

端点供应商接口

在某些情况下，Jakarta XML Web 服务需要能够在 XML 消息级别上操作。端点 提供程序 界面向实施它的 Web 服务提供此功能。

使用和访问端点

部署 Web 服务后，您可以使用 WSDL 创建组件存根，这是应用程序的基础。然后，您的应用可以访问端点来开展工作。

3.2.2. 开发和部署 Jakarta XML Web 服务端点

Jakarta XML Web 服务端点是一个服务器端组件，响应来自 Jakarta XML Web 服务客户端的请求，并为自身发布 WSDL 定义。

有关如何开发 Jakarta XML Web Services 端点应用程序的工作示例，请参见 JBoss EAP 附带的以下快速入门：

- jaxws-addressing
- jaxws-ejb
- jaxws-pojo
- jaxws-retail
- wsat-simple
- wsba-coordinator-completion-simple
- wsba-participant-completion-simple

开发要求

Web 服务必须满足 Jakarta XML Web Services API 和 Jakarta Web Services 元数据规范 2.1 规范的要求。

- 它包含 javax.jws.WebService 注释。
- 所有方法参数和返回类型都与 Jakarta XML Binding 2.3 规范 兼容。

以下是满足这些要求的 Web 服务实施示例。

示例：Web 服务实施

```
package org.jboss.quickstarts.ws.jaxws.samples.retail.profile;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
```

```
@Stateless

@WebService(
   name = "ProfileMgmt",
   targetNamespace = "http://org.jboss.ws/samples/retail/profile",
   serviceName = "ProfileMgmtService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public class ProfileMgmtBean {
   @WebMethod
   public DiscountResponse getCustomerDiscount(DiscountRequest request) {
      DiscountResponse dResponse = new DiscountResponse();
      dResponse.setCustomer(request.getCustomer());
      dResponse.setDiscount(10.00);
      return dResponse;
   }
}
```

以下是上例中 ProfileMgmtBean bean 使用的 DiscountRequest 类示例。注解包含在内。通常，Jakarta XML Binding 的默认值是正确的，不需要指定。

示例：DiscountRequest 类

```
package org.jboss.test.ws.jaxws.samples.retail.profile;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

import org.jboss.test.ws.jaxws.samples.retail.Customer;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(
  name = "discountRequest",
  namespace="http://org.jboss.ws/samples/retail/profile",
  propOrder = { "customer" }
)
public class DiscountRequest {

  protected Customer customer;

  public DiscountRequest() {
  }

  public DiscountRequest(Customer customer) {
    this.customer = customer;
  }
```

```
    public Customer getCustomer() {
      return customer;
    }

    public void setCustomer(Customer value) {
      this.customer = value;
    }

}
```

打包部署

实施类封装在 JAR 部署中。部署所需的任何元数据都从实施类和服务端点接口的注解中获取。您可以使用管理 CLI 或管理控制台部署 JAR，并且 HTTP 端点会自动创建。

下表显示了 Jakarta Enterprise Beans Web 服务的 JAR 部署结构示例：

```
$ jar -tf jaxws-samples-retail.jar
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtBean.class
org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

3.3. JAKARTA XML WEB SERVICES WEB SERVICE 客户端

3.3.1. 使用并访问 Jakarta XML Web 服务 Web 服务

在创建了 Web 服务端点（手动或使用 Jakarta XML Web 服务注释）后，您可以访问其 WSDL。此 WSDL 可用于创建将与 Web 服务通信的基本客户端应用。从发布的 WSDL 生成 Java 代码的过程称为使用 Web 服务的过程。这在以下阶段发生：

1. 创建客户端构件。

2. 构建服务存根.

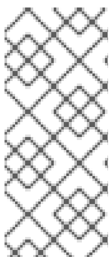
创建客户端工件

在创建客户端工件前，您需要创建 WSDL 合同。以下 WSDL 合同用于本节其余部分中介绍的示例。

以下示例依赖 ProfileMgmtService.wsdl 文件中具有此 WSDL 合同。

```
<definitions
    name='ProfileMgmtService'
    targetNamespace='http://org.jboss.ws/samples/retail/profile'
    xmlns='http://schemas.xmlsoap.org/wsdl/'
    xmlns:ns1='http://org.jboss.ws/samples/retail'
    xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
    xmlns:tns='http://org.jboss.ws/samples/retail/profile'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

   <types>

      <xs:schema targetNamespace='http://org.jboss.ws/samples/retail'
               version='1.0' xmlns:xs='http://www.w3.org/2001/XMLSchema'>
         <xs:complexType name='customer'>
            <xs:sequence>
               <xs:element minOccurs='0' name='creditCardDetails' type='xs:string'/>
               <xs:element minOccurs='0' name='firstName' type='xs:string'/>
               <xs:element minOccurs='0' name='lastName' type='xs:string'/>
            </xs:sequence>
         </xs:complexType>
      </xs:schema>

      <xs:schema
         targetNamespace='http://org.jboss.ws/samples/retail/profile'
         version='1.0'
         xmlns:ns1='http://org.jboss.ws/samples/retail'
         xmlns:tns='http://org.jboss.ws/samples/retail/profile'
         xmlns:xs='http://www.w3.org/2001/XMLSchema'>

         <xs:import namespace='http://org.jboss.ws/samples/retail'/>
         <xs:element name='getCustomerDiscount'
                  nillable='true' type='tns:discountRequest'/>
         <xs:element name='getCustomerDiscountResponse'
                  nillable='true' type='tns:discountResponse'/>
         <xs:complexType name='discountRequest'>
            <xs:sequence>
               <xs:element minOccurs='0' name='customer' type='ns1:customer'/>

            </xs:sequence>
         </xs:complexType>
         <xs:complexType name='discountResponse'>
            <xs:sequence>
               <xs:element minOccurs='0' name='customer' type='ns1:customer'/>
               <xs:element name='discount' type='xs:double'/>
            </xs:sequence>
         </xs:complexType>
      </xs:schema>
```

```
  </types>

  <message name='ProfileMgmt_getCustomerDiscount'>
    <part element='tns:getCustomerDiscount' name='getCustomerDiscount'/>
  </message>
  <message name='ProfileMgmt_getCustomerDiscountResponse'>
    <part element='tns:getCustomerDiscountResponse'
        name='getCustomerDiscountResponse'/>
  </message>
  <portType name='ProfileMgmt'>
    <operation name='getCustomerDiscount'
            parameterOrder='getCustomerDiscount'>

      <input message='tns:ProfileMgmt_getCustomerDiscount'/>
      <output message='tns:ProfileMgmt_getCustomerDiscountResponse'/>
    </operation>
  </portType>
  <binding name='ProfileMgmtBinding' type='tns:ProfileMgmt'>
    <soap:binding style='document'
              transport='http://schemas.xmlsoap.org/soap/http'/>
    <operation name='getCustomerDiscount'>
      <soap:operation soapAction=''/>
      <input>

        <soap:body use='literal'/>
      </input>
      <output>
        <soap:body use='literal'/>
      </output>
    </operation>
  </binding>
  <service name='ProfileMgmtService'>
    <port binding='tns:ProfileMgmtBinding' name='ProfileMgmtPort'>

      <!-- service address will be rewritten to actual one when WSDL is requested from running
server -->
      <soap:address location='http://SERVER:PORT/jaxws-retail/ProfileMgmtBean'/>
    </port>
  </service>
</definitions>
```



注意

如果您使用 Jakarta XML Web 服务注解来创建 Web 服务端点，则会自动生成 WSDL 合同，您只需要其 URL。您可以通过导航到 Runtime、选择适用的服务器、选择 Web 服务，然后选择 端点来找到此 URL。

wsconsume.sh 或 **wsconsume.bat** 工具用于使用抽象合同(WSDL)，并生成注解的 Java 类和定义它的可选源。该工具位于 *EAP_HOME*/bin/ 目录中。

```
$ ./wsconsume.sh --help
```

```
WSConsumeTask is a cmd line tool that generates portable JAX-WS artifacts from a WSDL file.

usage: org.jboss.ws.tools.cmd.WSConsume [options] <wsdl-url>

options:
    -h, --help                 Show this help message
    -b, --binding=<file>       One or more JAX-WS or Java XML Binding files
    -k, --keep                 Keep/Generate Java source
    -c  --catalog=<file>       Oasis XML Catalog file for entity resolution
    -p  --package=<name>       The target package for generated source
    -w  --wsdlLocation=<loc>   Value to use for @WebService.wsdlLocation
    -o, --output=<directory>   The directory to put generated artifacts
    -s, --source=<directory>   The directory to put Java source
    -t, --target=<2.0|2.1|2.2> The JAX-WS target
    -q, --quiet                Be somewhat more quiet
    -v, --verbose              Show full exception stack traces
    -l, --load-consumer        Load the consumer and exit (debug utility)
    -e, --extension            Enable SOAP 1.2 binding extension
    -a, --additionalHeaders    Enable processing of implicit SOAP headers
    -n, --nocompile            Do not compile generated sources
```

以下命令从 ProfileMgmtService .wsdl 文件生成输出中列出的 source. java 文件。源使用软件包的目录结构，该结构通过 -p 参数指定。

```
[user@host bin]$ wsconsume.sh -k -p org.jboss.test.ws.jaxws.samples.retail.profile
ProfileMgmtService.wsdl
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.java
output/org/jboss/test/ws/jaxws/samples/retail/profile/Customer.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountRequest.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/DiscountResponse.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ObjectFactory.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmt.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/ProfileMgmtService.class
output/org/jboss/test/ws/jaxws/samples/retail/profile/package-info.class
```

java 源 文件和已编译 .class 文件都生成到您运行 **命令的 目录中的 输出/ 目录中。**

表 3.3. wsconsume.sh创建的工件描述

File	描述
ProfileMgmt.java	服务端点接口.

File	描述
Customer.java	自定义数据类型.
discount.java	自定义数据类型.
ObjectFactory.java	Jakarta XML 绑定 XML 注册表.
package-info.java	Jakarta XML Binding 软件包注释.
ProfileMgmtService.java	服务工厂.

wsconsume 命令生成所有自定义数据类型（Jakarta XML Binding 注解的类）、服务端点接口和服务工厂类。这些工件用于构建 Web 服务客户端实施。

构建服务 Stub

Web 服务客户端使用服务存根来提取远程 Web 服务调用的详细信息。对于客户端应用，Web 服务调用类似于调用任何其他业务组件。在这种情况下，服务端点接口充当业务接口，服务工厂类则不将它构建为服务存根。

以下示例首先创建一个使用 WSDL 位置和服务名称的服务工厂。接下来，它将使用 **wsconsume** 创建的服务端点接口来构建服务存根。最后，可以像任何其他业务接口一样使用存根。

您可以在 JBoss EAP 管理控制台中找到您的端点的 WSDL URL。您可以通过导航到 **Runtime**、选择适用的服务器、选择 **Web 服务**，然后选择 端点来找到此 URL。

```
import javax.xml.ws.Service;
[...]
Service service = Service.create(
new URL("http://example.org/service?wsdl"),
new QName("MyService")
);
ProfileMgmt profileMgmt = service.getPort(ProfileMgmt.class);

// Use the service stub in your application
```

3.3.2. 开发 Jakarta XML Web 服务客户端应用程序

客户端与 Jakarta XML Web 服务端点通信并请求工作，该端点部署在 Java 企业版 7 容器中。有关下面提及的类、方法和其他实施详细信息，请查看 JBoss EAP 中包含的 Javadocs 捆绑包的相关部分。

概述

服务 是一种抽象，代表 WSDL 服务。WSDL 服务是相关端口的集合，各自包括绑定到特定协议的端口类型和特定的端点地址。

通常，当从现有的 WSDL 合同生成其余组件存根时，服务会生成。可以通过部署的端点的 WSDL URL 获取 WSDL 合同，也可以使用 *EAP_HOME*/bin/ 目录中的 wsprovide 工具从端点来源创建。

这种类型的用法被称为静态用例。在这种情况下，您要创建 Service 类的实例，它作为组件 stubs 之一创建。

您还可以使用 Service.create 方法手动创建服务。这称为动态用例。

使用
静态用例

Jakarta XML Web Services 客户端的静态用例假定您已有 WSDL 合同。这可以通过外部工具生成，或者在创建 Jakarta XML Web Services 端点时使用正确的 Jakarta XML Web Services 注解生成。

若要生成组件存根，可使用 *EAP_HOME*/bin 中包含的 wsconsume 工具。工具将 WSDL URL 或文件取为参数，并生成多个文件，这些文件结构在一个目录树中。代表您的 服务的 源和类文件分别命名为 _Service.java 和 _Service.class。

所生成的实施类具有两个公共构造器，一个没有参数，另一个具有两个参数。这两个参数分别代表 WSDL 位置(java.net.URL)和服务名称(javax.xml.namespace.QName)。

no-gument 构造器是最常使用的。在这种情况下，WSDL 位置和服务名称可在 WSDL 中找到。它们从 @WebServiceClient 注释隐式设置，该注释用于解密生成的类。

```
@WebServiceClient(name="StockQuoteService",
targetNamespace="http://example.com/stocks",
wsdlLocation="http://example.com/stocks.wsdl")
public class StockQuoteService extends javax.xml.ws.Service
{
  public StockQuoteService() {
    super(new URL("http://example.com/stocks.wsdl"), new
QName("http://example.com/stocks", "StockQuoteService"));
  }

  public StockQuoteService(String wsdlLocation, QName serviceName) {
    super(wsdlLocation, serviceName);
```

```
    }
    ...
}
```

有关如何从服务获取端口以及如何在该端口上调用操作的详情，请参阅 Dynamic Proxy。有关如何直接使用 XML 有效负载或使用整个 SOAP 消息的 XML 表示的详情，请参阅 Dispatch。

动态用例

在动态情况下，不会自动生成 stubs。相反，Web 服务客户端使用 Service.create 方法来创建 服务 实例。以下代码片段说明了此过程。

```
URL wsdlLocation = new URL("http://example.org/my.wsdl");
QName serviceName = new QName("http://example.org/sample", "MyService");
Service service = Service.create(wsdlLocation, serviceName);
```

处理程序 Resolver

Jakarta XML Web Services 为消息处理模块（称为处理程序）提供灵活的插件框架。这些处理程序扩展了 Jakarta XML Web 服务运行时系统的功能。Service 实例通过一对 getHandler Resolver 和 setHandlerResolver 方法提供对 Handler Resolver 的访问，它们可以在每个服务、每个端口或协议绑定的基础上配置一组处理程序。

当 服务 实例创建代理或 Dispatch 实例时，当前注册到该服务的处理程序解析器会创建所需的处理程序链。对为 Service 实例配置的处理器解析器的后续更改不会影响之前创建的代理或 Dispatch 实例上的处理程序。

执行器

服务 实例可以使用 java.util.concurrent.Executor 配置。Executor 调用应用请求的任何异步回调。setExecutor 和 getExecutor 方法可以修改和检索为服务配置的 Executor。

动态代理

动态代理是使用 服务 中提供的其中一个 getPort 方法 的客户端代理实例。portName 指定服务使用的 WSDL 端口的名称。serviceEndpointInterface 指定创建的动态代理实例支持的服务端点接口。

```
public <T> T getPort(QName portName, Class<T> serviceEndpointInterface)
public <T> T getPort(Class<T> serviceEndpointInterface)
```

服务端点接口通常使用 ws Consume 工具生成，该工具 解析 WSDL 并从中创建 Java 类。

还提供了返回端口的 typed 方法。这些方法也会返回实施 SEI 的动态代理。请参见以下示例。

```
@WebServiceClient(name = "TestEndpointService", targetNamespace =
"http://org.jboss.ws/wsref",
  wsdlLocation = "http://localhost.localdomain:8080/jaxws-samples-webserviceref?wsdl")

public class TestEndpointService extends Service {
   ...

   public TestEndpointService(URL wsdlLocation, QName serviceName) {
      super(wsdlLocation, serviceName);
   }

   @WebEndpoint(name = "TestEndpointPort")
   public TestEndpoint getTestEndpointPort() {
      return (TestEndpoint)super.getPort(TESTENDPOINTPORT, TestEndpoint.class);
   }
}
```

@WebServiceRef

@WebServiceRef 注释声明了对 Web 服务的引用。它遵循 JSR 250 中定义的 javax.annotation.Resource 注释所显示的资源模式。与这些注解对应的 Jakarta EE 遵循 Jakarta Annotations 1.3 规范。

- 您可以使用它来定义类型为生成的 Service 类的引用。在本例中，type 和 value 元素各自引用生成的 Service 类类型。此外，如果引用类型可以通过字段或方法声明推断，则该注解将应用到，则类型和值元素可能具有 Object.class 的默认值，但不是必须的。如果无法推断类型，则至少必须使用非默认值存在 type 元素。

- 您可以使用它来定义类型为 SEI 的引用。在这种情况下，如果可以从注解的字段或方法声明推断引用的类型，则 type 元素可能会（但不需要）使用默认值。但是，value 元素必须始终存在并引用生成的服务类类型，这是 javax.xml.ws.Service 的子类型。wsdlLocation 元素将覆盖所引用服务类的 @WebService 注释中指定的 WSDL 位置信息。

```
public class EJB3Client implements EJB3Remote
{
  @WebServiceRef
  public TestEndpointService service4;

  @WebServiceRef
  public TestEndpoint port3;
}
```

分配

XML Web 服务使用 XML 消息在端点和任何客户端之间进行通信，该端点部署在 Jakarta EE 容器中。XML 消息使用名为 Simple Object Access Protocol(SOAP)的 XML 语言。Jakarta XML Web 服务 API 为端点和客户端提供了能够发送和接收 SOAP 消息的机制。marshalling 是将 Java 对象转换为 SOAP XML 消息的过程。解压是将 SOAP XML 消息转换回 Java 对象的过程。

在某些情况下，您需要访问原始 SOAP 消息本身，而不是转换的结果。Dispatch 类提供此功能。分配以两种使用模式之一运行，由以下任一常量标识：

- javax.xml.ws.Service.Mode.MESSAGE - 此模式指示客户端应用直接使用特定于协议的消息结构。与 SOAP 协议绑定一起使用时，客户端应用程序直接与 SOAP 消息配合工作。

- javax.xml.ws.Service.Mode.PAYLOAD - 此模式使客户端能够处理载荷本身。例如，如果与 SOAP 协议绑定搭配使用，客户端应用程序将处理 SOAP 正文的内容，而不是整个 SOAP 消息。

分配 是一个低级别 API，要求客户端将消息或载荷构建为 XML，并严格遵守各个协议的标准以及对消息或有效载荷结构的详细了解。分配 是一个通用类，支持任何类型的消息或消息有效负载的输入和输出。

```
Service service = Service.create(wsdlURL, serviceName);
Dispatch dispatch = service.createDispatch(portName, StreamSource.class, Mode.PAYLOAD);

String payload = "<ns1:ping xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
dispatch.invokeOneWay(new StreamSource(new StringReader(payload)));

payload = "<ns1:feedback xmlns:ns1='http://oneway.samples.jaxws.ws.test.jboss.org/'/>";
Source retObj = (Source)dispatch.invoke(new StreamSource(new StringReader(payload)));
```

异步调用

BindingProvider 接口代表提供客户端可以使用的协议绑定的组件。它由代理实施，并由 Dispatch 接口扩展。

BindingProvider 实例可能会提供异步操作功能。异步操作调用在调用时与 BindingProvider 实例分离。操作完成后不会更新响应上下文。相反，可以使用 Response 接口提供单独的响应上下文。

```
public void testInvokeAsync() throws Exception {
  URL wsdlURL = new URL("http://" + getServerHost() + ":8080/jaxws-samples-asynchronous?wsdl");
  QName serviceName = new QName(targetNS, "TestEndpointService");
  Service service = Service.create(wsdlURL, serviceName);
  TestEndpoint port = service.getPort(TestEndpoint.class);
  Response response = port.echoAsync("Async");
  // access future
  String retStr = (String) response.get();
  assertEquals("Async", retStr);
}
```

@Oneway Invocations

@Oneway 注释表示给定 Web 方法采用输入消息，但不返回任何输出消息。通常，@Oneway 方法在执行业务方法之前，将控制线程返回到调用应用。

```
@WebService (name="PingEndpoint")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class PingEndpointImpl {
  private static String feedback;

  @WebMethod
  @Oneway
  public void ping() {
    log.info("ping");
    feedback = "ok";
  }

  @WebMethod
  public String feedback() {
    log.info("feedback");
    return feedback;
  }
}
```

超时配置

两种不同的属性控制 HTTP 连接的超时行为和等待接收消息的客户端的超时。第一个是 javax.xml.ws.client.connectionTimeout，第二个则是 javax.xml.ws.client.receiveTimeout。每个以毫秒为单位表示，正确语法如下所示：

```
public void testConfigureTimeout() throws Exception {
  //Set timeout until a connection is established
  ((BindingProvider)port).getRequestContext().put("javax.xml.ws.client.connectionTimeout",
"6000");

  //Set timeout until the response is received
  ((BindingProvider) port).getRequestContext().put("javax.xml.ws.client.receiveTimeout",
"1000");

  port.echo("testTimeout");
}
```

3.4. 配置 WEB 服务子系统

JBossWS 组件处理 Web 服务端点的处理，并通过 webservices 子系统提供给 JBoss EAP。子系统支持配置发布的端点地址和端点处理程序链。

服务器域和单机配置文件中提供了默认的 Web 服务子系统。它包含几个预定义的端点和客户端配置。

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
  <wsdl-host>${jboss.bind.address:127.0.0.1}</wsdl-host>
  <endpoint-config name="Standard-Endpoint-Config"/>
```

```
  <endpoint-config name="Recording-Endpoint-Config">
    <pre-handler-chain name="recording-handlers" protocol-bindings="##SOAP11_HTTP
##SOAP11_HTTP_MTOM ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
      <handler name="RecordingHandler"
class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
    </pre-handler-chain>
  </endpoint-config>
  <client-config name="Standard-Client-Config"/>
</subsystem>
```

3.4.1. 端点配置

JBossWS 支持预定义以及与端点实施关联的额外设置配置数据。预定义的端点配置可用于 Jakarta XML Web Services 客户端和 Jakarta XML Web Services 端点设置。端点配置可以包括 Jakarta XML Web Services 处理程序和键/值属性声明。此功能提供了一种便捷的方式，可将处理程序添加到 Web 服务端点和设置控制 JBossWS 和 Apache CXF 内部的键/值属性。

您可以通过 webservices 子系统定义指定的端点配置数据集合。每一端点配置在子系统内必须具有唯一的名称。然后，可以使用 org.jboss.ws.api.annotation.EndpointConfig 注解将端点配置分配给部署的应用中的 Jakarta XML Web Services 实施。有关 分配端点配置 的更多信息，请参阅分配配置。

默认 JBoss EAP 配置中有两个预定义的端点配置：

- standard-Endpoint-Config 是用于没有显式分配的端点配置的端点配置。
- Record-Endpoint-Config 是一个自定义端点配置示例，其中包含一个记录处理程序。

添加端点配置

您可以使用管理 CLI 添加新端点配置。

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config:add
```

配置端点配置

您可以使用管理 CLI 为端点配置添加键/值属性声明。

```
/subsystem=webservices/endpoint-config=Standard-Endpoint-
Config/property=PROPERTY_NAME:add(value=PROPERTY_VALUE)
```

您还可以为这些端点配置 处理程序链 和处理程序。

删除端点配置

您可以使用管理 CLI 删除端点配置。

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config:remove
```

3.4.2. 处理程序链

每个端点配置都可以与 PRE 或 POST 处理程序链关联。每个处理程序链可能包括 Jakarta XML Web 服务兼容处理程序，对消息执行额外的处理。对于出站消息，在使用标准 Jakarta XML Web 服务（如 @HandlerChain 注释）附加到端点的任何处理程序之前，执行 PRE 处理程序链处理程序。POST 处理程序链处理程序在常规端点处理程序后执行。对于入站消息，反之亦然。

服务器出站消息

```
Endpoint --> PRE Handlers --> Endpoint Handlers --> POST Handlers --> ... --> Client
```

服务器入站消息

```
Client --> ... --> POST Handlers --> Endpoint Handlers --> PRE Handlers --> Endpoint
```

添加处理程序链

您可以使用以下管理 CLI 命令将 POST 处理程序链添加到端点配置中：

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain:add
```

您可以使用以下管理 CLI 命令将 PRE 处理程序链添加到端点配置中：

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/pre-handler-chain=my-pre-handler-chain:add
```

配置处理程序链

使用 protocol-bindings 属性设置触发处理程序链启动的协议。

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain:write-attribute(name=protocol-bindings,value=##SOAP11_HTTP)
```

如需有关为处理程序链配置处理程序的信息，请参见 handlers 部分。

删除处理程序链

您可以使用管理 CLI 删除处理程序链。

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain:remove
```

3.4.3. 处理程序

Jakarta XML Web Services 处理程序添加到处理程序链中，并指定处理程序类的完全限定名称。部署端点时，会为每个引用部署创建一个该类的实例。部署类加载器或 org.jboss.as.webservices.server.integration 模块的 类加载程序必须能够加载处理程序类。

有关可用处理程序的列表，请参阅 Handler Javadocs。

添加处理程序

您可以使用以下管理 CLI 命令将处理程序添加到处理程序链：您必须提供处理程序的类名称。

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain/handler=my-handler:add(class="com.arjuna.webservices11.wsarj.handler.InstanceIdentifierInHandler")
```

配置处理程序

您可以使用管理 CLI 更新处理程序的类。

```
/subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain/handler=my-handler:add(class="org.jboss.ws.common.invocation.RecordingServerHandler")
```

删除处理程序

您可以使用管理 CLI 删除处理程序。

> /subsystem=webservices/endpoint-config=My-Endpoint-Config/post-handler-chain=my-post-handler-chain/handler=my-handler:

3.4.4. 发布的端点地址

支持重写 WSDL 协议中发布的端点的 <soap:address> 元素。此功能可用于控制为每个端点公告给客户端的服务器地址。

下表列出了可针对此功能配置的属性：

Name	描述
modify-wsdl-address	这个布尔值可启用并禁用地址重写功能。 当 **修改-wsdl-address** 设置为 **true** 时，**<soap:address>** 的内容是一个有效的 URL，JBossWS 使用 ws **dl-host 和 wsdl- port 或 wsdl- secure-port** 的值重写 URL。 当 **修改-wsdl-address** 设置为 **false** 且 **<soap:address>** 的内容是一个有效的 URL 时，JBossWS 不会重写该 URL。使用 **<soap:address>** URL。 当 **<soap:address> 的内容** 不是一个有效的 URL 时，JBossWS 会重写它，无论 **修改-wsdl-address** 的设置是什么。如果 **修改-wsdl-address** 设为 **true**，并且 **wsdl-host** 没有定义或明确设置为 **jbossws.undefined.host**，则使用 **<soap:address>** URL 的内容。JBossWS 在重写 **<soap:address>** 时使用请求者的主机。 当 **修改-wsdl-address** 没有定义 JBossWS 时，默认值为 **true**。
wsdl-host	用于重写 **<soap:address>** 的主机名或 IP 地址。如果将 **wsdl-host** 设置为 **jbossws.undefined.host**，JBossWS 在重写 **<soap:address>** 时使用请求者的主机。未定义 **wsdl-host** 时，JBossWS 使用默认值 **jbossws.undefined.host**。
wsdl-path-rewrite-rule	此字符串定义了 SED 替换命令，例如 **s/regexp/replacement/g**，即 JBossWS 针对服务器中发布的每个 **<soap:address>** URL 的路径组件执行。如果没有定义 **wsdl-path-rewrite-rule**，JBossWS 会保留每个 **<soap:address>** URL 的原始路径组件。当 **修改-wsdl-address** 设置为 **false** 时，此元素将被忽略。
wsdl-port	设置此属性以显式定义将用于重写 SOAP 地址的 HTTP 端口。否则，将通过查询已安装的 HTTP 连接器列表来标识 HTTP 端口。
wsdl-secure-port	设置此属性以明确定义将用于重写 SOAP 地址的 HTTPS 端口。否则，HTTPS 端口将通过查询已安装的 HTTPS 连接器列表来标识。

Name	描述
wsdl-uri-scheme	此属性明确设置用于重写 **<soap:address> 的 URI 方案**。有效值为 **http** 和 **https**。即使指定了传输保证，此配置也会覆盖通过处理端点计算的方案。**wsdl-port** 和 **wsdl-secure-port** 或其默认值会根据指定的方案使用提供的值。

您可以使用管理 CLI 更新这些属性。例如：

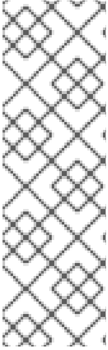
```
/subsystem=webservices:write-attribute(name=wsdl-uri-scheme, value=https)
```

3.4.5. 查看运行时信息

每一 Web 服务端点通过提供端点实施的部署公开。每个端点可以作为部署资源查询。每个 Web 服务端点指定一个 Web 上下文和 WSDL URL。您可以使用管理 CLI 或管理控制台访问这些运行时信息。

以下管理 CLI 命令显示了 jaxws-samples-handlerchain.war 部署中的 TestService 端点的详细信息。

```
/deployment="jaxws-samples-handlerchain.war"/subsystem=webservices/endpoint="jaxws-samples-handlerchain:TestService":read-resource(include-runtime=true)
{
    "outcome" => "success",
    "result" => {
        "average-processing-time" => 23L,
        "class" => "org.jboss.test.ws.jaxws.samples.handlerchain.EndpointImpl",
        "context" => "jaxws-samples-handlerchain",
        "fault-count" => 0L,
        "max-processing-time" => 23L,
        "min-processing-time" => 23L,
        "name" => "TestService",
        "request-count" => 1L,
        "response-count" => 1L,
        "total-processing-time" => 23L,
        "type" => "JAXWS_JSE",
        "wsdl-url" => "http://localhost:8080/jaxws-samples-handlerchain?wsdl"
    }
}
```



注意

在 read -resource 操作中使用 include-runtime=true 标志，会在结果中返回运行时统计信息。但是，默认禁用 Web 服务端点的收集。您可以使用以下管理 CLI 命令为 Web 服务端点启用统计信息。

```
/subsystem=webservices:write-attribute(name=statistics-enabled,value=true)
```

您还可以从管理控制台的 Runtime 选项卡中查看 Web 服务端点的运行时信息，方法是选择适用的服务器，选择 Web 服务， 然后选择端点。

3.5. 分配客户端和端点配置

可以通过以下方式分配客户端和端点配置：

- 通过注释（端点）或 API 编程为客户端明确分配.
- 从默认描述符自动分配配置。
- 自动分配容器的配置。

3.5.1. 显式配置分配

明确的配置分配适用于必须提前知道其端点或客户端的开发人员。配置来自应用部署中包含的描述符，或者包含在 webservices 子系统中。

3.5.1.1. 配置部署描述符

可以包含 Jakarta XML Web 服务客户端和端点实施的 Jakarta EE 存档还可以包含预定义的客户端和端点配置声明。给定存档的所有端点或客户端配置定义必须在一个部署描述符文件中提供，该文件必须是可在 EAP_HOME/docs/schema/schema/jbossws-jaxws-config_4_0.xsd 中找到的模式实施。可以在部署描述符文件中定义许多端点或客户端配置。每一配置必须具有在部署应用的服务器中唯一的名称。配置名称不能由应用外的端点或客户端实施引用。

示例：带有两个端点配置的描述符

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>org.jboss.test.ws.jaxws.jbws3282.Endpoint4Impl</config-name>
    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:handler>
          <javaee:handler-name>Log Handler</javaee:handler-name>
          <javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.LogHandler</javaee:handler-
class>
        </javaee:handler>
      </javaee:handler-chain>
    </pre-handler-chains>
    <post-handler-chains>
      <javaee:handler-chain>
        <javaee:handler>
          <javaee:handler-name>Routing Handler</javaee:handler-name>
          <javaee:handler-class>org.jboss.test.ws.jaxws.jbws3282.RoutingHandler</javaee:handler-
class>
        </javaee:handler>
      </javaee:handler-chain>
    </post-handler-chains>
  </endpoint-config>
  <endpoint-config>
    <config-name>EP6-config</config-name>
    <post-handler-chains>
      <javaee:handler-chain>
        <javaee:handler>
          <javaee:handler-name>Authorization Handler</javaee:handler-name>
          <javaee:handler-
class>org.jboss.test.ws.jaxws.jbws3282.AuthorizationHandler</javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>
    </post-handler-chains>
  </endpoint-config>
</jaxws-config>
```

同样，可以在描述符中指定客户端配置，该描述符仍在实施上述架构：

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <client-config>
    <config-name>Custom Client Config</config-name>
```

```
    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:handler>
          <javaee:handler-name>Routing Handler</javaee:handler-name>
          <javaee:handler-
class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</javaee:handler-class>
        </javaee:handler>
        <javaee:handler>
          <javaee:handler-name>Custom Handler</javaee:handler-name>
          <javaee:handler-
class>org.jboss.test.ws.jaxws.clientConfig.CustomHandler</javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>
    </pre-handler-chains>
  </client-config>
  <client-config>
    <config-name>Another Client Config</config-name>
    <post-handler-chains>
      <javaee:handler-chain>
        <javaee:handler>
          <javaee:handler-name>Routing Handler</javaee:handler-name>
          <javaee:handler-
class>org.jboss.test.ws.jaxws.clientConfig.RoutingHandler</javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>
    </post-handler-chains>
  </client-config>
</jaxws-config>
```

3.5.1.2. 应用程序服务器配置

JBoss EAP 允许在 webservices 子系统中声明 JBossWS 客户端和服务器预定义配置。因此，可以声明要添加到分配给给定配置的每个端点或客户端的链中的服务器范围的处理程序。

标准配置

默认情况下，在相同 JBoss EAP 实例和端点中运行的客户端被分配有标准配置。除非设置了不同的配置，否则将使用默认值。这样，管理员可以针对客户端和端点配置调整默认处理程序链。webservices 子系统中使用的默认客户端和端点配置的名称是 Standard-Client-Config 和 Standard-Endpoint-Config。

处理程序类

在设置服务器范围处理程序时，处理程序类需要通过各个 ws 部署类加载器来提供。因此，可能需要在将使用给定预定义配置的部署中指定正确的模块依赖项。确保部署中指定了正确的模块依赖项的一种方法是将依赖项添加到包含处理程序类的模块中，其中一个模块已自动设置为任何部署，如 org.jboss.ws.spi。

配置示例

示例：默认子系统配置

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
    <!-- ... -->
    <endpoint-config name="Standard-Endpoint-Config"/>
    <endpoint-config name="Recording-Endpoint-Config">
        <pre-handler-chain name="recording-handlers" protocol-bindings="##SOAP11_HTTP
##SOAP11_HTTP_MTOM ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
            <handler name="RecordingHandler"
class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
        </pre-handler-chain>
    </endpoint-config>
    <client-config name="Standard-Client-Config"/>
</subsystem>
```

部署特定 ws-security 端点设置的配置文件：

```
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="urn:jboss:jbossws-jaxws-
config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom WS-Security Endpoint</config-name>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-
value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</propert
y-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

JBoss EAP 默认配置被修改为默认为 SOAP 消息 schema-validation on:

```
<subsystem xmlns="urn:jboss:domain:webservices:2.0">
  <!-- ... -->
  <endpoint-config name="Standard-Endpoint-Config">
    <property name="schema-validation-enabled" value="true"/>
  </endpoint-config>
  <!-- ... -->
  <client-config name="Standard-Client-Config">
    <property name="schema-validation-enabled" value="true"/>
  </client-config>
</subsystem>
```

3.5.1.3. EndpointConfig Annotation

且配置可用于给定应用，则使用 org.jboss.ws.api.annotation.EndpointConfig 注解来分配端点配置到 Jakarta XML Web Services 端点实施。当您分配 webservices 子系统中定义的配置时，您只需要指定配置名称。当您分配应用中定义的配置时，您需要指定部署描述符的相对路径和配置名称。

示例： EndpointConfig 注解

```
@EndpointConfig(configFile = "WEB-INF/my-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
public class ServiceImpl implements ServiceIface {
  public String sayHello() {
    return "Secure Hello World!";
  }
}
```

3.5.1.4. Jakarta XML Web 服务功能

您还可以使用 org.jboss.ws.api.configuration.ClientConfig 功能设置 配置，该配置是 JBossWS 提供的 Jakarta XML Web 服务功能扩展。

```
import org.jboss.ws.api.configuration.ClientConfigFeature;

Service service = Service.create(wsdlURL, serviceName);

Endpoint port = service.getPort(Endpoint.class, new ClientConfigFeature("META-INF/my-
client-config.xml", "Custom Client Config"));
port.echo("Kermit");
```

您还可以通过传递 true 传递给 ClientConfigFeature 构造器来设置指定配置的属性。

```
Endpoint port = service.getPort(Endpoint.class, new ClientConfigFeature("META-INF/my-
client-config.xml", "Custom Client Config"), true);
```

在使用当前线程上下文类加载程序将其解析为资源后，JBossWS 解析了指定的配置文件。*EAP_HOME*/docs/schema/jbossws-jaxws-config_4_0.xsd 模式定义描述符内容，并包含在 jbossws-spi 构件中。

如果为配置文件传递 null，则将从当前容器配置中读取配置（如果可用）。

```
Endpoint port = service.getPort(Endpoint.class, new ClientConfigFeature(null, "Container
Custom Client Config"));
```

3.5.1.5. API 明确设置

或者，JBossWS API 附带可用于在构建客户端时分配配置的设备类。

处理程序

Jakarta XML Web 服务处理程序是从客户端配置中读取的，如下所示：

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);
BindingProvider bp = (BindingProvider)port;

ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client Config");
port.echo("Kermit");
```

您还可以使用 ClientConfigUtil 实用程序类来设置处理程序。

```
ClientConfigUtil.setConfigHandlers(bp, "META-INF/my-client-config.xml", "Custom Client
Config");
```

默认 ClientConfigurer 实施在作为使用当前线程上下文类加载程序的资源解析后，会解析指定的配置文件。EAP_HOME/docs/schema/jbossws-jaxws-config_4_0.xsd 模式定义了描述符内容，包含在 jbossws-spi 构件中。

如果为配置文件传递 null，则将从当前容器配置中读取配置（如果可用）。

```
ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigHandlers(bp, null, "Container Custom Client Config");
```

Properties

类似地，属性从客户端配置中读取，如下所示：

```
import org.jboss.ws.api.configuration.ClientConfigUtil;
import org.jboss.ws.api.configuration.ClientConfigurer;

Service service = Service.create(wsdlURL, serviceName);
Endpoint port = service.getPort(Endpoint.class);

ClientConfigUtil.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client
Config");
port.echo("Kermit");
```

您还可以使用 ClientConfigUtil 实用程序类来设置属性。

```
ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, "META-INF/my-client-config.xml", "Custom Client
Config");
```

默认 ClientConfigurer 实施在作为使用当前线程上下文类加载程序的资源解析后，会解析指定的配置文件。EAP_HOME/docs/schema/jbossws-jaxws-config_4_0.xsd 模式定义了描述符内容，包含在 jbossws-spi 构件中。

如果为配置文件传递 null，则将从当前容器配置中读取配置（如果可用）。

```
ClientConfigurer configurer = ClientConfigUtil.resolveClientConfigurer();
configurer.setConfigProperties(port, null, "Container Custom Client Config");
```

3.5.2. 从默认描述符自动配置

在某些情况下，应用开发人员可能不知道需要用于其客户端和端点实施的配置。在其他情况下，可能不接受 JBossWS API 的显式使用，因为它是编译时间依赖关系。为应对这样的情形，JBossWS 允许在其根目录中包括应用的默认客户端、jaxws-client -config.xml 和 端点 jaxws-endpoint -config.xml。在未指定配置文件名称时，这些将被解析为获取配置。

```
<config-file>WEB-INF/jaxws-endpoint-config.xml</config-file>
```

如果没有指定配置名称，JBossWS 会自动查找名为 的配置：

- 端点实施类的完全限定名称(FQN)，适用于 Jakarta XML Web 服务端点。
- 服务端点接口的 FQN，用于 Jakarta XML Web Services 客户端。

没有为 Dispatch 客户端选择自动配置名称。

例如，一个端点实施类 org.foo.bar.EndpointImpl （没有显式设置预定义的配置）将导致 JBossWS 在应用部署 的根目录中查找 jaxws-endpoint-config.xml 描述符中的 org.foo. bar.EndpointImpl。同样，在客户端，实施 org.foo.bar.Endpoint 接口的客户端代理将从 jaxws-client-config .xml 描述符中名为 configuration 的 org.foo.bar. Endpoint 进行设置读取。

3.5.3. 从容器自动分配配置

在未提供显式配置并且默认描述符不可用或不包含相关配置时，JBossWS 将回退为从容器获取预定义配置。此行为为管理员提供了对 Jakarta XML Web Services 客户端和端点设置的额外控制，因为可以独立于部署的应用管理容器。

JBossWS 访问显式命名配置的 webservices 子系统。使用的默认配置名称有：

- 端点实施类的完全限定名称，适用于 Jakarta XML Web 服务端点。
- 服务端点接口的完全限定名称，适用于 Jakarta XML Web Services 客户端。

分配 客户端不会自动配置。如果没有找到使用以上名称计算的配置，则 Standard-Client-Config 和 Standard-Endpoint-Config 配置分别用于客户端和端点。

3.6. 为 WEB 服务应用设置模块依赖项

JBoss EAP Web 服务以一组模块和库的形式交付，包括 org.jboss.as.webservices.* 和 org.jboss.ws.* 模块。您应该不需要更改这些模块。

使用 JBoss EAP 时，您无法直接使用 JBossWS 实施类，除非将依赖关系明确设置为对应的模块。您可以声明您要添加到部署中的模块依赖项。

只要有 webservices 子系统可用，都默认可用 JBossWS API。您可以使用它们而不为这些模块创建显式依赖项声明。

3.6.1. 使用 MANIFEST.MF

若要配置部署依赖项，请将它们添加到 MANIFEST.MF 文件中。例如：

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services export,foo.bar
```

此 MANIFEST.MF 文件声明对 org.jboss.ws.cxf.jbossws-cxf-client 和 foo.bar 模块的 依赖关系。有关在 MANIFEST.MF 文件中声明依赖关系的更多信息（包括 导出 和服务 选项），请参阅 JBoss EAP *开发指南中* 向 MANIFEST.MF 添加依赖性配置。

在端点和处理程序上使用注释时，如 Apache CXF 端点和处理程序，请在清单文件中添加正确的模块依赖项。如果您跳过此步骤，您的注解不会被使用，并完全被静默忽略。

3.6.1.1. 使用 Jakarta XML Binding

要在容器中运行的客户端或端点中成功直接使用 Jakarta XML Binding 上下文，请设置 Jakarta XML Binding 实施。例如，设置以下依赖项：

```
Dependencies: com.sun.xml.bind services export
```

3.6.1.2. 使用 Apache CXF

要使用 Apache CXF API 和实施类，请将依赖项添加到 org.apache.cxf (API)模块或 org.apache.cxf.impl （实施）模块。例如：

```
Dependencies: org.apache.cxf services
```

其依赖性纯粹是 Apache CXF，没有任何 JBossWS 自定义或附加扩展。因此，客户端聚合模块可用于您可能需要的所有 Web 服务依赖项。

3.6.1.3. 客户端 Web 服务聚合模块

当您要使用所有 Web 服务功能和功能时，您可以将依赖项设置为便捷的客户端模块。例如：

```
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client services
```

需要 services 选项通过加载 JBossWS 特定的类来启用所有 JBossWS 功能。在声明 org.jboss.ws.cxf .jbossws-cxf 和 org. apache.cxf 模块的依赖关系时， 几乎始终需要 services 选项。选项影响通过 服务 API 加载类，这是用于连线大多数 JBossWS 组件和 Apache CXF 总线扩展的类。

3.6.1.4. 注解扫描

应用服务器使用注释索引来检测用户部署中的 Jakarta XML Web 服务端点。为属于不同模块的类声明 Web 服务端点时，如在 web.xml 描述符中引用它，请使用 注释 类型依赖项。如果没有这种依赖关系，您的端点将被忽略，因为它们不会显示为 webservices 子系统的带注释类。

```
Dependencies: my.org annotations
```

3.6.2. 使用 jboss-deployment-structure.xml

在某些情况下，在 MANIFEST.MF 文件中设置模块依赖关系的便捷方法可能不起作用。例如，在从给定模块依赖项导入和导出特定资源时，在 MANIFEST.MF 文件中设置依赖关系时无法正常工作。在这些情况下，将 jboss-deployment-structure.xml 描述符文件添加到您的部署中，并在其中设置模块依赖项。

有关使用 jboss-deployment-structure.xml 的更多信息，请参阅 JBoss EAP *开发指南* 中的 向 jboss-deployment-structure.xml 添加依赖配置。

3.7. 配置 HTTP 超时

HTTP 会话超时定义 HTTP 会话被视为无效的时间段，因为指定期间内没有活动。

HTTP 会话超时可以根据优先顺序在以下位置进行配置：

1. **Application（应用程序）**

   您可以通过向 文件中添加以下配置，在应用的 web.xml 配置文件中定义 HTTP 会话超时：这个值以分钟为单位。

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

如果您修改了 WAR 文件，请重新部署应用。如果您浏览了 WAR 文件，则不需要进一步的操作，因为 JBoss EAP 自动取消部署并重新部署应用。

2. **Server**

   您可以使用以下管理 CLI 命令，在 undertow 子系统中设置默认的 HTTP 会话超时：这个值以分钟为单位。

   ```
   /subsystem=undertow/servlet-container=default:write-attribute(name=default-session-timeout,value=30)
   ```

3. **默认**

   默认的 HTTP 会话超时为 30 分钟。

3.8. 保护 JAKARTA XML WEB 服务

WS-Security 提供了一种方法，可以保护您的服务超越传输级别协议，如 HTTPS。通过很多标准，如 WS-Security 标准中定义的标头，您可以：

- 在服务之间传递身份验证令牌.
- 加密消息或消息部分。
- 签署消息.
- 时间戳消息.

WS-Security利用了公钥和私钥加密。使用公钥加密时，用户拥有一对公钥和私钥。这些是使用大量主要编号和一个关键功能生成的。

这些键在数学上是相关的，但不能互相派生。通过这些密钥，我们可以加密消息。例如，如果 Scott 想要向 Adam 发送消息，他可以使用他的公钥加密消息。然后 Adam 可以使用他的私钥解密此消息。只有 Adam 可以解密此消息，因为他是唯一具有私钥的消息。

也可以对消息进行签名。这样，您可以确保消息的真实性。如果 Adam 想要向 Scott 发送一封消息，Scott 希望确保它来自 Adam，Adam 可以使用他的私钥签署该消息。然后，Scott 可以使用他的公钥验证消息是否来自 Adam。

3.8.1. 应用 Web 服务安全性(WS-Security)

Web 服务支持许多需要 WS 安全功能的真实场景。这些场景包括签名和加密支持，通过 X509 证书支持，通过用户名令牌进行身份验证和授权，以及 WS-SecurityPolicy 规范涵盖的所有 WS-Security 配置。

对于其他 WS-* 功能，WS-Security 功能的核心通过 Apache CXF 引擎提供。此外，JBossWS 集成还添加了一些配置增强功能，以简化启用 WS-Security 端点的设置。

3.8.1.1. Apache CXF WS-Security 实施

Apache CXF 具有支持多种配置且易于扩展的 WS-Security 模块。

系统基于为低级安全操作委派给 Apache WSS4J *的拦截器*。可以通过不同的方式配置拦截器，可以通过 Spring 配置文件或直接使用 Apache CXF 客户端 API 配置拦截器。

最新版本的 Apache CXF 引入了对 WS-SecurityPolicy 的支持，该支持旨在通过策略将大部分安全配置移至服务合同中，以便客户能够从该协议中几乎可以完全自动配置。这样，用户不需要手动处理所需拦截器的配置和安装；Apache CXF WS-Policy 引擎在内部需要处理。

3.8.1.2. WS-安全政策支持

WS-SecurityPolicy 描述了与给定 WSDL 合同中公告的服务进行安全通信所需的操作。WSDL 绑定和操作引用 WS-Policy 片段与与服务交互的安全要求。WS-SecurityPolicy 规范允许指定诸如非对称和对称密钥、使用传输(HTTPS)进行加密、哪些部分或标头加密或签名、是否要包含时间戳、是否使用派生密钥或其他方面。

但是 WS-SecurityPolicy 不涵盖一些必需的配置元素，因为它们不是公开的或已发布端点合同的一部

分。其中包括密钥存储位置以及用户名和密码等内容。Apache CXF 允许通过 Spring XML 描述符或使用客户端 API 或注释来配置这些元素。

表 3.4. 支持的配置属性

配置属性	描述
ws-security.username	用于 **UsernameToken 策略** 断言的用户名。
ws-security.password	用于 **UsernameToken 策略** 断言的密码。如果没有指定，则会调用回调处理程序。
ws-security.callback-handler	WSS4J 安全 **回调Handler**，它将用于检索密钥存储和 **用户名令牌的密码**。
ws-security.signature.properties	包含用于配置签名密钥存储和加密对象的 WSS4J 属性的属性 file/对象。
ws-security.encryption.properties	包含用于配置加密密钥存储和加密对象的 WSS4J 属性的属性 file/对象。
ws-security.signature.username	要使用的签名密钥存储中的密钥的用户名或别名。如果未指定，它将使用属性文件中设置的默认别名。如果没有设置该密钥，并且密钥存储仅包含一个密钥，则将使用该密钥。
ws-security.encryption.username	要使用的加密密钥存储中的密钥的用户名或别名。如果未指定，它将使用属性文件中设置的默认别名。如果没有设置该密钥，并且密钥存储仅包含一个密钥，则将使用该密钥。对于 Web 服务提供商，**useReqSigCert** 关键字可用于接受（加密）任何公钥存在于服务的信任存储中的客户端（在 **ws-security.encryption.properties 中定义**）。
ws-security.signature.crypto	这可以指向完整的 WSS4J **Crypto** 对象，而不是指定签名属性。这样可以更轻松地对加密信息进行编程配置。
ws-security.encryption.crypto	这可以指向完整的 WSS4J **Crypto** 对象，而不是指定加密属性。这样可以更轻松地对加密信息进行编程配置。
ws-security.enable.streaming	启用 WS-Security 消息的流传输（基于 StAX）处理.

3.8.2. ws-Trust

WS-Trust 是一种 Web 服务规范，用于定义对 WS-Security 的扩展。它是在分布式系统中实施安全性的一般框架。该标准基于集中式安全令牌服务(STS)，该服务能够验证客户端并发行包含各种身份验证和

授权数据的令牌。该规范描述了用于安全性令牌规范、交换和验证的协议。以下规格在 WS-Trust 架构中扮演重要角色：

- WS-SecurityPolicy 1.2
- SAML 2.0
- 用户名令牌配置集
- X.509 令牌配置集
- SAML 令牌配置集
- Kerberos 令牌配置集

WS-Trust 扩展满足了跨越多个域且需要共享安全密钥的应用程序的需求。这可以通过提供基于标准的可信第三方 Web 服务(STS)来代理 Web 服务请求者和 Web 服务提供商之间的信任关系。此架构还通过提供此信息的常用位置，减轻了需要更改凭据的服务更新的难度。STS 是请求者和提供程序从中检索并验证安全令牌的常用访问点。

WS-Trust 规范有三个主要组件：

- 用于发布、续订和验证安全令牌的安全令牌服务(STS)。
- 安全令牌请求和响应的消息格式。
- 密钥交换机制。

下面的部分解释了基本的 WS-Trust 场景。有关高级场景，请参阅 高级 WS-Trust 场景。

3.8.2.1. 场景：基本 WS-Trust

在本节中，我们提供了一个基本的 WS-Trust 场景示例。它包含一个 Web 服务请求器(ws-requester)、Web 服务提供商(ws-provider)和安全令牌服务(STS)。

ws-provider 需要从指定的 STS 发布的 SAML 2.0 令牌由 ws-requester 使用非 对称绑定呈现。这些通信要求在 ws-provider 的 WSDL 中声明。STS 要求 ws-requester 凭据通过对称绑定在 WSS UsernameToken 格式请求中提供。提供来自 STS 的响应包含 SAML 2.0 令牌。这些通信要求在 STS 的 WSDL 中声明。

1. ws-requester 联系 ws-provider 并使用其 WSDL。在找到安全令牌签发者要求时，ws-requester 会创建并配置 STSClient，使其包含生成有效请求所需的信息。

2. STSClient 与 STS 联系并使用其 WSDL。发现安全策略。STSClient 创建并发送具有适当凭据的身份验证请求。

3. STS 验证凭据。

4. 作为响应，STS 签发了一个安全令牌，它提供 ws-requester 通过 STS 进行身份验证的证明。

5. STSClient 向 ws-provider 呈现一条安全令牌消息。

6. ws-provider 验证令牌是否由 STS 签发，因此证明 ws-requester 已通过 STS 成功验证了令牌。

7. ws-provider 执行请求的服务，并将结果返回到 ws-requester。

3.8.2.2. Apache CXF 支持

Apache CXF 是一个开源、功能全面的 Web 服务框架。JBossWS 开源项目将 JBoss Web Services(JBossWS)堆栈与 Apache CXF 项目模块集成，以提供 WS-Trust 和其他 Jakarta XML Web 服务功能。这种集成有助于轻松部署 Apache CXF STS 实施。Apache CXF API 还提供 STSClient 实用程序，以协助 Web 服务请求者与其 STS 通信。

3.8.3. 安全令牌服务(STS)

安全令牌服务(STS)是 WS-Trust 规范的核心。它是一种基于标准的验证和授权机制。STS 是基于令牌格式、命名空间或信任边界的 WS-Trust 规范协议用于发布、交换和验证安全令牌的一种实施。STS 是一个 Web 服务，充当可信第三方，以代理 Web 服务请求者和 Web 服务提供商之间的信任关系。它是请求者和提供商信任的常用接入点，可提供可互操作的安全令牌。它免除了请求者和供应商之间直接关系的需求。STS 有助于确保跨域和不同平台之间的互操作性，因为它是基于标准的身份验证机制。

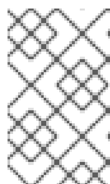
STS 的 WSDL 合同定义了其他应用程序和进程如何与之交互。特别是，WSDL 定义了 WS-Trust 和 WS-Security 策略，请求者必须遵守这些策略才能与 STS 的端点成功通信。Web 服务请求者使用 STS 的 WSDL，并且在 STSClient 实用程序的帮助下，生成符合所声明安全策略的消息请求并将其提交到 STS 端点。STS 验证请求并返回适当的响应。

3.8.3.1. 配置 PicketLink WS-Trust 安全令牌服务(STS)

PicketLink STS 提供构建 Apache CXF 安全令牌服务实施的替代方案的选项。您还可以使用 PicketLink 为 Web 应用配置 SAML SSO。有关使用 PicketLink 配置 SAML SSO 的更多详细信息，请参阅 *如何使用 SAML v2 设置 SSO*。

要将应用程序设置为 PicketLink WS-Trust STS，必须执行以下步骤：

1. 为 WS-Trust STS 应用程序创建安全域。
2. 为 WS-Trust STS 应用程序配置 web.xml 文件。
3. 为 WS-Trust STS 应用程序配置身份验证器。
4. 声明 WS-Trust STS 应用所需的依赖关系。
5. 配置 WS-Trust STS 应用程序的 web-service 部分。
6. 为 WS-Trust STS 应用创建和配置 picket link.xml 文件。



注意

在创建和部署应用之前，应创建和配置安全域。

3.8.3.1.1. 为 STS 创建安全域

STS 根据提供的凭据处理主体的身份验证，并根据结果发布正确的安全令牌。这要求通过安全域配置身份存储。创建此安全域和身份存储的唯一要求是其已正确定义了身份验证和授权机制。这意味着，可以利用许多不同的身份存储（如属性文件、数据库和 LDAP）以及它们关联的登录模块来支持 STS 应用。如需有关安全域的更多信息，请参阅 JBoss EAP 安全架构 *文档中* 的安全 *域* 部分。

在以下示例中，使用一个简单的 UsersRoles 登录模块，该模块使用身份服务的属性文件。

用于创建安全域的 CLI 命令

```
/subsystem=security/security-domain=sts:add(cache-type=default)
```

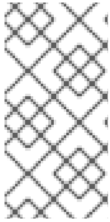
```
/subsystem=security/security-domain=sts/authentication=classic:add
```

```
/subsystem=security/security-domain=sts/authentication=classic/login-
module=UsersRoles:add(code=UsersRoles,flag=required,module-options=
[usersProperties=${jboss.server.config.dir}/sts-
users.properties,rolesProperties=${jboss.server.config.dir}/sts-roles.properties])
```

```
reload
```

生成 XML

```
<security-domain name="sts" cache-type="default">
 <authentication>
  <login-module code="UsersRoles" flag="required">
   <module-option name="usersProperties" value="${jboss.server.config.dir}/sts-users.properties"/>
   <module-option name="rolesProperties" value="${jboss.server.config.dir}/sts-roles.properties"/>
  </login-module>
 </authentication>
</security-domain>
```



注意

显示的管理 CLI 命令假定您在运行 JBoss EAP 单机服务器。有关将管理 CLI 用于 JBoss EAP 受管域的更多详细信息，请参见 JBoss EAP *管理 CLI 指南*。

属性文件

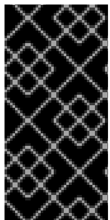
UsersRoles login 模块利用属性文件来存储用户/密码和用户/角色信息。有关 UsersRoles 模块的更多详情，请参阅 JBoss EAP *登录模块参考*。在本例中，属性文件包含以下内容：

示例： sts-users.properties 文件

```
Eric=samplePass
Alan=samplePass
```

示例： sts-roles.properties File

```
Eric=All
Alan=
```



重要

您还需要创建用于签名和加密安全令牌的密钥存储。配置 picket link.xml 文件时将使用此密钥存储。

3.8.3.1.2. 为 STS 配置 web.xml 文件

STS 的 web.xml 文件应包含以下内容：

- 一个 <servlet>，用于启用 STS 功能和 <servlet-mapping> 来 映射其 URL。

- 带有 < web-resource-collection> 的 <security-constraint >，其中包含一个 <url-pattern>，它映射到安全区域的 URL 模式。另外，<security-constraint> 也可以包含 <auth-constraint> 来 替代允许的角色。

- 为 BASIC 身份验证 配置 <login-config>。

- 如果在 <auth-constraint> 中指定了任何角色，这些角色应在 <security-role> 中定义。

web.xml 文件示例：

```
<web-app>
  <!-- Define STS servlet -->
  <servlet>
    <servlet-name>STS-servlet</servlet-name>
    <servlet-class>com.example.sts.PicketLinkSTService</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>STS-servlet</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
  <!-- Define a security constraint that requires the All role to access resources -->
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>STS</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>All</role-name>
    </auth-constraint>
  </security-constraint>
  <!-- Define the Login Configuration for this Application -->
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>STS Realm</realm-name>
  </login-config>
  <!-- Security roles referenced by this web application -->
  <security-role>
    <description>The role that is required to log in to the IDP Application</description>
    <role-name>All</role-name>
  </security-role>
</web-app>
```

3.8.3.1.3. 为 STS 配置 Authenticator

身份验证器负责用户的身份验证来发行和验证安全令牌。身份验证器通过定义要在验证和授权主体时使用的安全域进行配置。

jboss-web.xml 文件应具有以下内容：

- <security-domain>，用于指定用于身份验证和授权的安全域。

示例： jboss-web.xml 文件

```
<jboss-web>
  <security-domain>sts</security-domain>
  <context-root>SecureTokenService</context-root>
</jboss-web>
```

3.8.3.1.4. 声明 STS 的 Necessary 依赖项

Web 应用充当 STS 要求在 jboss-deployment-structure.xml 文件中定义依赖项，以便能够找到 org.picketlink 类。由于 JBoss EAP 提供所有必需的 org.picketlink 和相关类，应用只需将其声明为依赖项即可使用它们。

示例：使用 jboss-deployment-structure.xml 进行 Declare 依赖项

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.picketlink"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

3.8.3.1.5. 配置 STS 的 Web-Service 端口

Web 应用充当 STS 要求您定义一个 Web 服务，供客户端调用来获取其安全令牌。这要求您在

WSDL 中定义一个名为 PicketLinkSTS 的服务名称，以及名为 PicketLinkSTSPort 的端口。但是，您可以更改 SOAP 地址来更好地反映您的目标部署环境。

示例： PicketLinkSTS.wsdl 文件

```
<?xml version="1.0"?>
<wsdl:definitions name="PicketLinkSTS" targetNamespace="urn:picketlink:identity-federation:sts"
 xmlns:tns="urn:picketlink:identity-federation:sts"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
 xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <wsdl:types>
    <xs:schema targetNamespace="urn:picketlink:identity-federation:sts"
      xmlns:tns="urn:picketlink:identity-federation:sts"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      version="1.0" elementFormDefault="qualified">
      <xs:element name="MessageBody">
        <xs:complexType>
          <xs:sequence>
            <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="RequestSecurityToken">
    <wsdl:part name="rstMessage" element="tns:MessageBody"/>
  </wsdl:message>
  <wsdl:message name="RequestSecurityTokenResponse">
    <wsdl:part name="rstrMessage" element="tns:MessageBody"/>
  </wsdl:message>
  <wsdl:portType name="SecureTokenService">
    <wsdl:operation name="IssueToken">
      <wsdl:input wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
message="tns:RequestSecurityToken"/>
      <wsdl:output wsap10:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Issue"
message="tns:RequestSecurityTokenResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="STSBinding" type="tns:SecureTokenService">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="IssueToken">
      <soap12:operation soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
style="document"/>
      <wsdl:input>
        <soap12:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
```

```
  </wsdl:binding>
  <wsdl:service name="PicketLinkSTS">
    <wsdl:port name="PicketLinkSTSPort" binding="tns:STSBinding">
      <soap12:address location="http://localhost:8080/SecureTokenService/PicketLinkSTS"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

此外，您的 web 服务需要一个类来使用 WSDL：

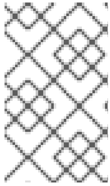
示例： PicketLinkSTS 类

```
@WebServiceProvider(serviceName = "PicketLinkSTS", portName = "PicketLinkSTSPort",
targetNamespace = "urn:picketlink:identity-federation:sts", wsdlLocation = "WEB-
INF/wsdl/PicketLinkSTS.wsdl")
@ServiceMode(value = Service.Mode.MESSAGE)
public class PicketLinkSTService extends PicketLinkSTS {
    private static Logger log = Logger.getLogger(PicketLinkSTService.class.getName());

    @Resource
    public void setWSC(WebServiceContext wctx) {
        log.debug("Setting WebServiceContext = " + wctx);
        this.context = wctx;
    }
}
```

3.8.3.1.6. 为 STS 创建和配置 picketlink.xml 文件

picketlink.xml 文件负责身份验证器的行为，并在应用的启动时载入。

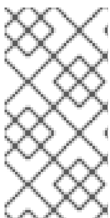
JBoss EAP 安全令牌服务定义多个提供扩展点的接口。可以在 中插入实施，并且可以利用配置为某些属性指定默认值。与 *如何设置带有 SAML v2 的 SSO* 中的 IDP 和 SP 配置类似，所有 STS 配置都在部署的应用的 picketlink.xml 文件中指定。以下是可以在 picket link.xml 文件中配置的元素：



注意

在以下文本中，服务提供商引用需要其客户端出示安全令牌的 Web 服务。

- <PicketLinkSTS> ：这是根元素。它定义了允许 STS 管理员设置以下属性的一些属性：
    - STSName ：代表安全令牌服务名称的字符串。如果没有指定，则使用默认的 PicketLinkSTS 值。
    - TokenTimeout ：令牌生命周期值（以秒为单位）。如果没有指定，则使用默认值 3600 （一小时）。
    - EncryptToken ：指定是否要加密发布的令牌的布尔值。默认值为 false。
- <KeyProvider> ：此元素及其所有子元素都用于配置 PicketLink STS 用于签名和加密令牌的密钥存储。本节中已配置密钥存储位置、密码以及签名（私钥）别名和密码等属性。
- <TokenProviders> ：本节指定 TokenProvider 实施，必须用来处理每种类型的安全令牌。在示例中，我们有两个提供程序 - 处理 SAMLV1.1 类型的令牌，一个处理 SAMLV2.0 类型的令牌。WSTrustRequestHandler 调用 STSConfiguration 的 getProviderForTokenType(String type) 方法，以获取对相应的 TokenProvider 的引用。
- <ServiceProviders> ：本节指定每个服务提供商必须使用的令牌类型，即需要安全令牌的 Web 服务。当 WS-Trust 请求不包含令牌类型时，WSTrustRequestHandler 必须使用服务提供商端点来找出必须发布的令牌类型。



注意

在配置 PicketLink 时，建议使用 POST 绑定，因为它提供增强的安全性，且不会在 URL 参数内传递响应。

示例： picketlink.xml 配置文件

```
<!DOCTYPE PicketLinkSTS>
```

```
<PicketLinkSTS xmlns="urn:picketlink:federation:config:2.1"
            STSName="PicketLinkSTS" TokenTimeout="7200" EncryptToken="false">
   <KeyProvider
          ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
      <Auth Key="KeyStoreURL" Value="sts_keystore.jks"/>
      <Auth Key="KeyStorePass" Value="testpass"/>
      <Auth Key="SigningKeyAlias" Value="sts"/>
      <Auth Key="SigningKeyPass" Value="keypass"/>
      <ValidatingAlias Key="http://services.testcorp.org/provider1"
                    Value="service1"/>
   </KeyProvider>
   <TokenProviders>
      <TokenProvider

ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML11TokenProvider"
            TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1"
            TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:1.0:assertion"/>
      <TokenProvider

ProviderClass="org.picketlink.identity.federation.core.wstrust.plugins.saml.SAML20TokenProvider"
            TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV2.0"
            TokenElement="Assertion" TokenElementNS="urn:oasis:names:tc:SAML:2.0:assertion"/>
   </TokenProviders>
   <ServiceProviders>
      <ServiceProvider Endpoint="http://services.testcorp.org/provider1"
                    TokenType="http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0"
                    TruststoreAlias="service1"/>
   </ServiceProviders>
</PicketLinkSTS>
```

默认情况下，opt etlink.xml 文件位于 STS Web 应用的 WEB-INF/classes 目录中。PicketLink 配置文件也可以从文件系统加载。要从文件系统加载 PicketLink 配置文件，必须将它命名为 picket link-sts.xml，并位于 ${user.home}/picketlink-store/sts/ 目录中。

3.8.3.2. 使用带有客户端的 WS-Trust 安全令牌服务(STS)

若要将客户端配置为从 STS 获取安全令牌，您需要使用 org.picketlink.identity.federation.api.wstrust.WSTrustClient 类以连接到 STS 并请求签发令牌。

首先，您需要实例化客户端：

示例：创建 WSTrustClient

```
WSTrustClient client = new WSTrustClient("PicketLinkSTS", "PicketLinkSTSPort",
    "http://localhost:8080/SecureTokenService/PicketLinkSTS",
    new SecurityInfo(username, password));
```

接下来，您需要使用 WSTrustClient 来要求提供令牌，如 SAML 断言：

示例：包含 Assertion

```
org.w3c.dom.Element assertion = null;
try {
  assertion = client.issueToken(SAMLUtil.SAML2_TOKEN_TYPE);
} catch (WSTrustException wse) {
  System.out.println("Unable to issue assertion: " + wse.getMessage());
  wse.printStackTrace();
}
```

断言后，可以通过两种方式将它包含在 并通过 SOAP 消息发送：

- 客户端可以将 SAML2 Assertion 推送到密钥 org.picketlink.trust.saml.assertion 下的 SOAP MessageContext 中。例如：

  ```
  bindingProvider.getRequestContext().put(SAML2Constants.SAML2_ASSERTION_PRO
  PERTY, assertion);
  ```

- SAML2 Assertion 在安全上下文中作为 JAAS 主题的一部分提供。如果 JAAS 与 PicketLink STS 登录模块的使用进行了交互，则会出现此情况。

3.8.3.3. STS 客户端池

警告

JBoss EAP 不支持 STS 客户端池功能。

STS 客户端池是一种允许您在服务器上配置 STS 客户端池的功能，从而消除了 STS 客户端创建可能出现的瓶颈。客户端池可用于需要 STS 客户端来获取 SAML 票据的登录模块。包括：

- org.picketlink.identity.federation.core.wstrust.auth.STSIssuingLoginModule
- org.picketlink.identity.federation.core.wstrust.auth.STSValidatingLoginModule
- org.picketlink.trust.jbossws.jaas.JBWSTokenIssuingLoginModule

每个登录模块的池中的默认客户端数量都使用 initialNumberOfClients 登录模块选项进行配置。

org.picketlink.identity.federation.bindings.stspool.STSClientPoolFactory 类为应用提供客户端池功能。

使用 STSClientPoolFactory

STS 客户端通过其 STSClientConfig 配置作为键插入到子池。要将 STS 客户端插入到子池中，您需要获取 STSClientPool 实例，然后根据配置初始化子池。（可选）您可以在初始化池时指定 STS 客户端的初始数量，或者您可以依赖默认编号。

示例：将 STS 客户端插入 Subpool

```
final STSClientPool pool = STSClientPoolFactory.getPoolInstance();
pool.createPool(20, stsClientConfig);
final STSClient client = pool.getClient(stsClientConfig);
```

使用完客户端后，您可以通过调用 returnClient（） 方法将它返回到池。

示例：将 STS 客户端返回到 Subpool

```
pool.returnClient();
```

示例：检查是否使用给定配置检查子池是否存在

```
if (! pool.configExists(stsClientConfig) {
   pool.createPool(stsClientConfig);
}
```

如果启用了 picketlink-federation 子系统，则为部署创建的所有客户端池都会在取消部署过程中自动销毁。手动销毁池：

示例：手动销毁子池

```
pool.destroyPool(stsClientConfig);
```

3.8.4. 将经过身份验证的身份传播到 Jakarta Enterprise Beans 子系统

webservices 子系统包含一个适配器，允许您配置 Elytron 安全域以使用注释或部署描述符保护 Web 服务端点的安全性。

启用 Elytron 安全性后，可将 JAAS 主题或主体推送到 Apache CXF 端点的 SecurityContext，将经过身份验证的身份传播到 Jakarta Enterprise Beans 容器。

以下是如何使用 Apache CXF 拦截器将验证信息传播到 Jakarta Enterprise Beans 容器的示例。

```
public class PropagateSecurityInterceptor extends WSS4JInInterceptor {
    public PropagateSecurityInterceptor() {
      super();
      getAfter().add(PolicyBasedWSS4JInInterceptor.class.getName());
   }
   @Override
   public void handleMessage(SoapMessage message) throws Fault {
      ...
      final Endpoint endpoint = message.getExchange().get(Endpoint.class);
      final SecurityDomainContext securityDomainContext =
endpoint.getSecurityDomainContext();
      //push subject principal retrieved from CXF to ElytronSecurityDomainContext
      securityDomainContext.pushSubjectContext(subject, principal, null)
      }
    }
```

3.9. JAKARTA XML WEB 服务日志记录

您可以使用 Jakarta XML Web Services 处理程序或 Apache CXF 日志记录拦截器 来处理入站和出站消息的日志。

3.9.1. 使用 Jakarta XML Web 服务处理程序

您可以配置 Jakarta XML Web Services 处理程序，以记录传递给它的消息。此方法可移植，因为处理程序可以通过使用 @HandlerChain Jakarta XML Web 服务注释编程方式添加到所需的客户端和端点。

预定义的客户端和端点配置机制允许您将日志记录处理程序添加到任何客户端和端点组合中，或者仅添加到部分客户端和端点。若要仅将日志记录处理程序添加到某些客户端或端点，可使用 @EndpointConfig 注释和 JBossWS API。

org.jboss.ws.api.annotation.EndpointConfig 注解用于分配端点配置到 Jakarta XML Web Services 端点实施。在分配 webservices 子系统中定义的配置时，仅指定配置名称。在分配应用中定义的配置时，必须指定部署描述符的相对路径和配置名称。

3.9.2. 使用 Apache CXF Logging Interceptors

Apache CXF 还附带日志记录拦截器，可用于记录消息到控制台、客户端日志文件或服务器日志文件。这些拦截器可以通过多种方式添加到客户端、端点和总线中，包括：

- **系统属性**

  将 org.apache.cxf.logging.enabled 系统属性 设置为 true 会导致日志记录拦截器添加到 JVM 上创建的任何总线实例中。您也可以将 system 属性设置为 com，以 输出格式良好的 XML 输出。您可以使用以下管理 CLI 命令来设置此系统属性：

  ```
  /system-property=org.apache.cxf.logging.enabled:add(value=true)
  ```

- **手动添加拦截器**

  可以使用 Apache CXF 注释 @org.apache.cxf.interceptor.InInterceptors 和 @org.apache.cxf.interceptor.OutInterceptors，有选择地将日志记录拦截器添加到端点。在客户端一侧通过编程方式向客户端或总线添加新的日志记录拦截器实例来实现同样的结果。

3.10. 启用 WEB 服务寻址(WS-ADDRESSING)

Web 服务寻址或 WS-Addressing 提供了一种不传输机制来寻址 Web 服务及其相关消息。若要启用 WS-Addressing，您必须将 @Addressing 注释添加到 Web 服务端点，然后配置客户端以进行访问。

以下示例假定您的应用具有现有的 Jakarta XML Web 服务服务和客户端配置。有关完整的工作示例，请参见 JBoss EAP 附带的 jaxws-addressing 快速入门。

1. 将 @Addressing 注释添加到应用的 Jakarta XML Web Services 端点代码。

   示例：带有 @Addressing Annotation 的 Jakarta XML Web Services Endpoint

   ```
   package org.jboss.quickstarts.ws.jaxws.samples.wsa;

   import org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface;

   import javax.jws.WebService;
   import javax.xml.ws.soap.Addressing;

   @WebService(
      portName = "AddressingServicePort",
      serviceName = "AddressingService",
      wsdlLocation = "WEB-INF/wsdl/AddressingService.wsdl",
      targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
      endpointInterface = "org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface")
   @Addressing(enabled = true, required = true)
   ```

```
public class ServiceImpl implements ServiceIface {
   public String sayHello() {
      return "Hello World!";
   }
}
```

2. 更新 Jakarta XML Web Services 客户端代码，以配置 WS-Addressing。

示例：为 WS-Addressing 配置的 Jakarta XML Web 服务客户端

```
package org.jboss.quickstarts.ws.jaxws.samples.wsa;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.AddressingFeature;

public final class AddressingClient {
   private static final String serviceURL =
      "http://localhost:8080/jaxws-addressing/AddressingService";

   public static void main(String[] args) throws Exception {
      // construct proxy
      QName serviceName =
         new QName("http://www.jboss.org/jbossws/ws-extensions/wsaddressing",
            "AddressingService");
      URL wsdlURL = new URL(serviceURL + "?wsdl");
      Service service = Service.create(wsdlURL, serviceName);
      org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface proxy =
         (org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface)
service.getPort(org.jboss.quickstarts.ws.jaxws.samples.wsa.ServiceIface.class,
            new AddressingFeature());
      // invoke method
      System.out.println(proxy.sayHello());
   }
}
```

客户端和端点现在使用 WS-Addressing 进行通信。

3.11. 启用 WEB 服务可靠的消息传递

Web 服务可靠的消息传递（WS 可靠消息传递）在 Apache CXF 内部实施。组拦截器与可靠的消息传递协议的低级别要求交互。

要启用 WS-可靠的消息传递，请完成以下步骤之一：

- 使用 WSDL 合同，指定正确的 WS 可靠的消息传递策略和声明，或两者。
- 手动添加和配置可靠的消息传递拦截器。
- 在可选 CXF Spring XML 描述符中指定可靠的消息传递策略。
- 在可选 CXF Spring XML 描述符中指定 Apache CXF 可靠的消息传递功能。

第一种方法是唯一可移植的方法，它依赖于 Apache CXF WS-Policy 引擎。其他专有方法允许对 WS 可靠的消息策略中未涵盖的协议方面进行精细配置。

3.12. 指定 WEB 服务策略

Web 服务策略(WS-Policy)依赖于 Apache CXF WS-Policy 框架。这个框架符合以下规格：

- Web 服务政策1.5 - Framework
- Web 服务政策 1.5 - 附件

您可以以不同的方式使用策略，包括：

- 在 WSDL 合同中添加策略断言，并让运行时使用断言并相应地执行行为。
- 使用 CXF 注解或功能指定端点策略附加。

- 使用 Apache CXF 策略框架定义自定义断言并完成其他任务。

3.13. APACHE CXF 集成

JBossWS 在 JBoss EAP 上提供的所有 Jakarta XML Web 服务功能目前均通过将 JBossWS 堆栈与大多数 Apache CXF 项目模块进行正确集成来提供。

Apache CXF 是一个开源服务框架。它允许使用前端编程 API（包括 Jakarta XML Web 服务）来构建和开发服务，这些服务通过 HTTP 和 Jakarta Messaging 等各种传输使用 SOAP 和 XML/HTTP 等协议。

JBossWS 和 Apache CXF 之间的集成层主要用于：

- 允许在 JBoss EAP 上使用标准 Web 服务 API，包括 Jakarta XML Web 服务；这是在内部利用 Apache CXF 执行，无需用户处理；
- 允许在 JBoss EAP 上使用 Apache CXF 高级功能（包括 WS-*），而无需用户处理、设置或关注在此类容器中运行所需的集成步骤。

为了支持这些目标，JBossWS 与 Apache CXF 集成支持 JBossWS 端点部署机制，并在 Apache CXF 基础上提供多种内部定制功能。

有关 Apache CXF 架构的更深入详情，请参阅 Apache CXF 官方文档。

3.13.1. 服务器端集成自定义

JBossWS 服务器与 Apache CXF 的端集成可为提供的 Web 服务部署在内部创建正确的 Apache CXF 结构。如果部署包含多个端点，则它们将存在于同一 Apache CXF 总线中，这与其他部署的总线实例分开。

虽然 JBossWS 为服务器端的大多数 Apache CXF 配置选项设置了明智的默认值，但用户可能希望微调为其部署创建的总线实例；jboss-webservices.xml 描述符可用于部署级别自定义。

3.13.1.1. Deployment Descriptor Properties

jboss-webservices.xml 描述符可用于提供属性值。

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  ...
  <property>
    <name>...</name>
    <value>...</value>
  </property>
  ...
</webservices>
```

JBossWS 与 Apache CXF 集成附带一组允许的属性名称，以控制 Apache CXF 内部。

3.13.1.2. WorkQueue 配置

Apache CXF 使用 WorkQueue 实例来处理一些操作，例如 @Oneway 请求处理。WorkQueueManager 安装在总线中作为扩展名，并允许添加或删除队列以及控制现有的队列。

在服务器端，可以使用 jboss-webservices.xml 中的 cxf.queue.<queue-name>.* 属性来提供队列。例如，您可以使用 cxf.queue.default.maxQueueSize 属性来配置默认 工作队列的最大队列大小。在部署时，JBossWS 集成可将 AutomaticWorkQueueImpl 的新实例添加到当前配置的 WorkQueueManager 中。以下属性用于填写 AutomaticWorkQueueImpl 构造器 参数：

表 3.5. AutomaticWorkQueueImpl Constructor Properties

属性	默认值
cxf.queue.<queue-name>.maxQueueSize	256
cxf.queue.<queue-name>.initialThreads	0
cxf.queue.<queue-name>.highWaterMark	25
cxf.queue.<queue-name>.lowWaterMark	5
cxf.queue.<queue-name>.dequeueTimeout	120000

3.13.1.3. 策略替代选择器

Apache CXF 策略引擎支持不同的策略来处理策略替代方案。JBossWS 集成当前默认为 MaximalAlternativeSelector，但仍允许使用 jboss-webservices.xml 文件中的 cxf.policy.alternativeSelector 属性来设置不同的选择器实施。

3.13.1.4. MBean 管理

Apache CXF 允许您管理安装到 JBoss EAP MBean 服务器的 MBean 对象。您可以通过 jboss-webservices.xml 文件中的 cxf.management.enabled 属性，在部署基础上启用此功能。您还可以使用 cxf.management.installResponseTimeInterceptors 属性来控制 CXF 响应时间拦截器的安装。在启用 MBean 管理时，这些拦截器会被默认添加，但在某些情况下可能不需要这些拦截器。

示例： jboss-webservices.xml 文件中的 MBean 管理

```
<webservices xmlns="http://www.jboss.com/xml/ns/javaee" version="1.2">
  <property>
    <name>cxf.management.enabled</name>
    <value>true</value>
  </property>
  <property>
    <name>cxf.management.installResponseTimeInterceptors</name>
    <value>false</value>
  </property>
</webservices>
```

3.13.1.5. 架构验证

Apache CXF 包含了在客户端和服务器端验证传入和传出 SOAP 信息的功能。验证针对端点 WSDL 合同（服务器端）或用于构建服务代理（客户端一侧）中的 WSDL 合同中的相关模式进行。

您可以使用以下任一方式启用 schema 验证：

- JBoss EAP 服务器配置中：

  例如，以下管理 CLI 命令为默认的 Standard-Endpoint-Config 端点配置启用架构验证。

  ```
  /subsystem=webservices/endpoint-config=Standard-Endpoint-Config/property=schema-validation-enabled:add(value=true)
  ```

- 在预定义的客户端或端点配置文件中。

  您可以通过在引用的 配置文件中将 schema-validation-enabled 属性设置为 true，将容器

内运行的任何端点或客户端与 JBossWS 预定义 配置关联。

- 以编程方式在客户端一侧.

  在客户端，您可以编程方式启用模式验证。例如：

```
((BindingProvider)proxy).getRequestContext().put("schema-validation-enabled", true);
```

- 在服务器端使用 @org.apache.cxf.annotations.SchemaValidation 注释。

  在服务器端，您可以使用 @org.apache.cxf.annotations.SchemaValidation 注释。例如：

```
import javax.jws.WebService;
import org.apache.cxf.annotations.SchemaValidation;

@WebService(...)
@SchemaValidation
public class ValidatingHelloImpl implements Hello {
   ...
}
```

3.13.1.6. Apache CXF Interceptors

jboss-webservices.xml 描述符可指定 cxf.interceptors.in 和 cxf.interceptors.out 属性。这些属性允许您将声明拦截器附加到为提供部署而创建的 总线 实例。

示例： jboss-webservices.xml 文件

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
  xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.2"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.interceptors.in</name>
    <value>org.jboss.test.ws.jaxws.cxf.interceptors.BusInterceptor</value>
  </property>
  <property>
    <name>cxf.interceptors.out</name>
```

```
    <value>org.jboss.test.ws.jaxws.cxf.interceptors.BusCounterInterceptor</value>
  </property>
</webservices>
```

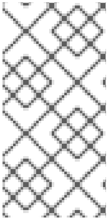
您可以使用以下方法之一声明拦截器：

- 端点类上的注释用法，如 @org.apache.cxf.interceptor.InInterceptor 或 @org.apache.cxf.interceptor.OutInterceptor.
- 通过 org.apache.cxf.interceptor.InterceptorProvider 接口直接在客户端使用 API。
- JBossWS 描述符使用.

由于 JBoss EAP 中不再支持 Spring 集成，因此 JBossWS 集成使用 jaxws-endpoint-config.xml 描述符文件，以避免对实际客户端或端点代码进行修改。您可以在预定义的客户端和端点配置中声明拦截器，方法是为 cxf.interceptors. in 和 cxf.interceptors. out 属性指定拦截器类名称列表。

示例： jaxws-endpoint-config.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointImpl</config-name>
    <property>
      <property-name>cxf.interceptors.in</property-name>
      <property-
value>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointInterceptor,org.jboss.test.ws.jaxws.cxf
.interceptors.FooInterceptor</property-value>
    </property>
    <property>
      <property-name>cxf.interceptors.out</property-name>
      <property-
value>org.jboss.test.ws.jaxws.cxf.interceptors.EndpointCounterInterceptor</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```



注意

每个指定的拦截器类的新实例将添加到分配了该配置的客户端或端点。拦截器类必须具有无参数构造器。

3.13.1.7. Apache CXF 功能

jboss-webservices.xml 描述符可指定 cxf.features 属性。此属性允许您声明要附加到属于为服务部署创建的 总线 实例的任何端点。

示例： jboss-webservices.xml 文件

```
<?xml version="1.1" encoding="UTF-8"?>
<webservices
  xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="1.2"
  xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee">

  <property>
    <name>cxf.features</name>
    <value>org.apache.cxf.feature.FastInfosetFeature</value>
  </property>
</webservices>
```

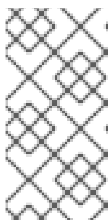
您可以使用以下方法之一声明功能：

- 端点类上的注释用法，如 @org.apache.cxf.feature.Features.
- 通过 org.apache.cxf.feature.AbstractFeature 类的扩展直接在客户端使用 API。
- JBossWS 描述符使用.

由于 JBoss EAP 中不再支持 Spring 集成，因此 JBossWS 集成添加了额外的描述符，即基于 jaxws-endpoint-config.xml 文件的方法，以避免对实际客户端或端点代码进行修改。您可以通过指定 cxf.features 属性的功能类名称列表，在预定义的客户端和端点配置中声明功能。

示例： jaxws-endpoint-config.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Custom FI Config</config-name>
    <property>
      <property-name>cxf.features</property-name>
      <property-value>org.apache.cxf.feature.FastInfosetFeature</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```



注意

每个指定功能类的新实例将添加到分配配置的客户端或端点。功能类必须具有无参数构造器。

3.13.1.8. 属性 - 生成 Bean 创建

Apache CXF Interceptors 和 Apache CXF 功能 部分解释了如何通过客户端或端点预定义配置或 jboss-webservices.xml 描述符中的属性声明 CXF 拦截器和功能。通过仅获取指定功能或拦截器类名称，容器尝试使用类默认构造器创建 bean 实例。这会对功能或拦截器配置设置限制，除非提供了 vanilla CXF 类的自定义扩展，并在最终使用超级构造器之前通过默认的构造器设置属性。

为解决此问题，JBossWS 集成在从属性构建时附带配置简单 Bean 层次结构的机制。属性可以具有 bean 引用值，它是以 ## 开头的字符串。属性引用键用于指定 bean 类名称和各个属性的值。

例如，以下属性会导致堆栈安装两个功能实例：

键	值
cxf.features	##foo, ##bar
##foo	org.jboss.Foo
##foo.par	34
##bar	org.jboss.Bar
##bar.color	Blue

以下代码可创建相同的结果：

```
import org.Bar;
import org.Foo;
...
Foo foo = new Foo();
foo.setPar(34);
Bar bar = new Bar();
bar.setColor("blue");
```

这种机制假定类是具有正确 getter（）和 setter（）方法 的有效 be an。通过检查类定义，值对象将转换为正确的原语类型。也可以配置嵌套的 Bean。

附录 A. 参考资料

A.1. JAKARTA RESTFUL WEB SERVICES/RESTEASY ANNOTATIONS

表 A.1. Jakarta RESTful Web Services/RESTEasy Annotations

注解	使用
cache	自动设置响应 **Cache-Control** 标头。
ClientInterceptor	将拦截器识别为客户端拦截器。
ContentEncoding	元注释，用于指定要通过注解的注解来应用的 **Content-Encoding**。
上下文	允许您指定 **javax.ws.rs.core.HttpHeaders**，**javax.ws.rs.core.UriInfo**，**javax.ws.rs.core.Request**，**javax.servlet.HttpServletRequest**，**javax.servlet.HttpServletResponse**, 和 **javax.ws.rs.core.SecurityContext** 对象.
CookieParam	允许您在方法调用中指定 HTTP 请求 cookie 或对象表示的值。
DecorateTypes	必须放在 **DecoratorProcessor** 类上，以指定受支持的类型。
decorator	将 meta-annotation 放置到触发解码的另一个注解上。
DefaultValue	可以与其他 **@*Param** 注释结合使用，以在 HTTP 请求项目不存在时定义默认值。
删除	表示方法响应 HTTP **DELETE** 请求的注释。
DoNotUseJAXBProvider	当您不想使用 Jakarta XML Binding **MessageBodyReader**/**Writer** 时，将其放入类或参数上，但具有更具体的提供程序来托管该类型。
encoded	可用于类、方法或参数。默认情况下，注入 **@PathParam** 和 **@QueryParams** 将被解码。通过添加 **@Encoded** 注释，这些 params 的值以编码形式提供。
表单	这可用作传入/传出请求/响应的值对象。
格式	使用缩进和换行格式化 XML 输出。这是一个 Jakarta XML Binding Decorator。

注解	使用
GET	表示方法响应 HTTP **GET** 请求的注释。
IgnoreMediaTypes	放置在类型、方法、参数或字段上，告知 Jakarta RESTful Web Services 不将 Jakarta XML Binding provider 用于特定的介质类型
ImageWriterParams	资源组可用于将参数传递给 **IIOImageProvider** 的注释。
映射	**JSONConfig**。
MultipartForm	这可用作 multipart/form-data MIME 类型的传入/传出请求/响应的值对象。
noCache	设置 **nocache** 的 **Cache-Control** 响应标头。
NoJackson	当您不希望触发 Jackson 提供程序时，将放置到类、参数、字段或方法。
PartType	写出 List 或 Map 作为多部分 /* **类型时，必须与多部分** 供应商一起使用。
路径	这必须存在于类或资源方法中。如果两者中都存在，则资源方法的相对路径是类和方法的串联。
PathParam	允许您将变量 URI 路径片段映射到方法调用。
POST	表示方法响应 HTTP **POST** 请求的注释。
优先级	用于指示使用类的顺序的注释。使用带有较低值的整数参数表示更高的优先级。
提供者	在供应商扫描阶段，将可发现的类 标记为 Jakarta RESTful Web Services 运行时作为提供商的类。
PUT	表示方法响应 HTTP **PUT** 请求的注释。
QueryParam	允许您将 URI 查询字符串参数或 URL 形式编码参数映射到方法调用。
ServerInterceptor	将拦截器识别为服务器端拦截器。
StringParameterUnmarshallerBinder	要放入另一个注解上的 meta-annotation，该注解会触发 **StringParameterUnmarshaller** 应用到基于字符串的注解注入器。

注解	使用
风格表	指定 XML 样式表标头。
wrapped	在您要托管或解封 Jakarta XML Binding 对象的集合或数组时，将此放在方法或参数中。
WrappedMap	当您想要托管或解封 Jakarta XML Binding 对象的映射时，将其放入方法或参数中。
XmlHeader	为返回的文档设置 XML 标头。
XmlNsMap	**JSONToXml.**
XopWithMultipartRelated	此注释可用于处理/复制传入/传出的 XOP 消息（打包为多部分/相关）到/来自 Jakarta XML Binding 注解的对象。

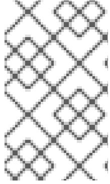
A.2. RESTEASY 配置参数

表 A.2. 元素

选项名称	默认值	描述
resteasy.servlet.mapping.prefix	没有默认	如果 Resteasy servlet-mapping 的 URL-pattern 不是 /*。
resteasy.scan	false	为 **@Provider 和 Jakarta RESTful Web Services 资源类（如 @Path**、@**GET**、@**POST**）自动扫描 **WEB-INF/lib JARs 和 WEB-INF/classes** 目录，并进行注册。
resteasy.scan.providers	false	扫描 **@Provider** 类并注册它们。
resteasy.scan.resources	false	扫描 Jakarta RESTful Web Services 资源类.
resteasy.providers	没有默认	以逗号分隔的列表，其中包含您希望 **注册的完全限定 @Provider** 类名称。
resteasy.use.builtin.providers	true	是否要注册默认的内置 **@Provider** 类.
resteasy.resources	没有默认	您要注册的完全限定的 Jakarta RESTful Web Services 资源类名称的逗号分隔列表。

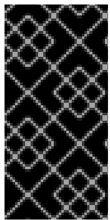
选项名称	默认值	描述
resteasy.jndi.resources	没有默认	个以逗号分隔的 JNDI 名称列表，引用您要注册为 Jakarta RESTful Web Services 资源的对象。
javax.ws.rs.Application	没有默认	以 spec 可移植的方式将应用程序 **类的** 完全限定名称到 bootstrap。
resteasy.media.type.mappings	没有默认	将文件名扩展名（如 **.xml** 或 **.txt**）映射到介质类型，取代 **Accept** 标头的需求。客户端无法使用 **Accept** 标头来选择表示法（如浏览器）。您可以在 **WEB-INF/web.xml** 文件中使用 **resteasy.media.type.mappings** 和 **resteasy.language.mappings** 配置此项。
resteasy.language.mappings	没有默认	将文件名扩展（如. **en** 或 **.fr**）映射到语言，不再需要 **Accept-Language** 标头。当客户端无法使用 **Accept-Language** 标头来选择语言（如浏览器）时使用。
resteasy.document.expand.entity.references	false	是否扩展外部实体，还是将它们替换为空字符串。在 JBoss EAP 中，此参数默认为 **false**，因此它将它们替换为空字符串。
resteasy.document.secure.processing.feature	true	在处理 **org.w3c.dom.Document** 文档和 Jakarta XML Binding 对象表示中施加安全限制。
resteasy.document.secure.disableDTDs	true	禁止 **org.w3c.dom.Document** 文档中的 DTDs 和 Jakarta XML Binding 对象表示。
resteasy.wider.request.matching	true	关闭 Jakarta RESTful Web Services 规范中定义的类级表达式过滤，而是根据每个 Jakarta RESTful Web Services 方法的完整表达式匹配。
resteasy.use.container.form.params	true	使用 **HttpServletRequest.getParameterMap（）** 方法来获取表单参数。如果您在 servlet 过滤器中调用此方法或者在过滤器内消耗输入流，请使用这个切换。

选项名称	默认值	描述
resteasy.add.charset	true	如果资源方法返回了 **文本/*** 或 **应用程序/xml*** 介质类型，但没有显式 charset，RESTEasy 会将 **charset=UTF-8** 添加到返回的内容类型标头中。请注意，charset 在此情况下默认为 UTF-8，独立于此参数的设置。



注意

这些参数在 WEB-INF/web.xml 文件中配置。



重要

在 Servlet 3.0 容器中，web .xml 文件中的 resteasy.scan. * 配置被忽略，所有 Jakarta RESTful Web Services 注解的组件都会被自动扫描。

例如，javax.ws.rs.Application 参数是在 servlet 配置 的 init-param 中配置的：

```
<servlet>
    <servlet-name>Resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>org.jboss.resteasy.utils.TestApplication</param-value>
    </init-param>
</servlet>
```

例如，resteasy.document.expand.entity.references 在 context-param 中配置：

```
<context-param>
    <param-name>resteasy.document.expand.entity.references</param-name>
    <param-value>true</param-value>
</context-param>
```

警告

更改以下 RESTEasy 参数的默认值可能会导致 RESTEasy 应用程序受到 XXE 攻击的影响：

- resteasy.document.expand.entity.references
- resteasy.document.secure.processing.feature
- resteasy.document.secure.disableDTDs

A.3. RESTEASY JAVASCRIPT API 参数

表 A.3. 参数属性

属性	默认值	描述
$entity		作为 **PUT** 发送的实体，**POST 请求**.
$contentType		作为 **Content-Type** 标头发送的正文实体的 MIME 类型。由 **@Consumes** 注释决定.
$accepts	*/*	接受的 MIME 类型作为 **Accept** 标头发送。由 **@Provides 注释决定**.
$callback		设置为异步调**用的**函数（httpCode、xmlHttpRequest、value）。如果不存在，则调用将同步并返回值。
$apiURL		设置为 Jakarta RESTful Web Services 端点的基础 URI，而不包括最后一个斜杠。
$username		如果设置了用户名和密码，它们将用于请求的凭据。
$password		如果设置了用户名和密码，它们将用于请求的凭据。

A.4. RESTEAS.REQUEST 类成员

表 A.4. resteas.Request 类

成员	描述
execute(callback)	使用当前对象中设置的所有信息执行请求。该值传递到可选参数回调，不返回。
setAccepts(acceptHeader)	设置 **Accept** 请求标头。默认值为 ***/***。
setCredentials(username, password)	设置请求凭证。
setEntity(entity)	设置请求实体。
setContentType(contentTypeHeader)	设置 **Content-Type** 请求标头。
setURI(uri)	设置请求 URI。这应该是绝对 URI。
setMethod(method)	设置请求方法。默认值为 **GET**。
setAsync(async)	控制请求是否异步。默认值为 **true**。
addCookie(name, value)	执行请求时，设置当前文档中给定的 Cookie。这会在浏览器中持久存在。
addQueryParameter(name, value)	将查询参数添加到 URI 查询部分。
addMatrixParameter(name, value)	将矩阵参数（路径参数）添加到请求 URI 的最后一个路径片段中。
addHeader(name, value)	添加请求标头。
addForm（名称，值）	添加表单。
addFormParameter(name, value)	添加形式参数。

A.5. RESTEASY 异步作业服务配置参数

下表详细介绍了异步作业服务的可配置 context-params。可以在 web.xml 文件中配置这些参数。

表 A.5. 配置参数

参数	描述
resteasy.async.job.service.max.job.results	在任意时间可在内存中保存的作业结果数。默认值为 **100**。
resteasy.async.job.service.max.wait	客户端正在查询作业时，最长等待时间。默认值为 **300000**。

参数	描述
resteasy.async.job.service.thread.pool.size	运行作业的后台线程池大小。默认值为 **100**。
resteasy.async.job.service.base.path	设置作业 URI 的基本路径。默认值为 **/asynch/jobs**。

```
<web-app>
   <context-param>
      <param-name>resteasy.async.job.service.enabled</param-name>
      <param-value>true</param-value>
   </context-param>

   <context-param>
      <param-name>resteasy.async.job.service.max.job.results</param-name>
      <param-value>100</param-value>
   </context-param>
   <context-param>
      <param-name>resteasy.async.job.service.max.wait</param-name>
      <param-value>300000</param-value>
   </context-param>
   <context-param>
      <param-name>resteasy.async.job.service.thread.pool.size</param-name>
      <param-value>100</param-value>
   </context-param>
   <context-param>
      <param-name>resteasy.async.job.service.base.path</param-name>
      <param-value>/asynch/jobs</param-value>
   </context-param>

   <listener>
      <listener-class>
         org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
      </listener-class>
   </listener>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>
         org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

</web-app>
```

A.6. JAKARTA XML WEB 服务工具

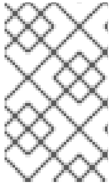
wsconsume

ws Consume 是随 JBoss EAP 提供的命令行工具，它使用 WSDL 并生成可移植的 Jakarta XML Web 服务服务和客户端工件。

使用

wsconsume 工具位于 EAP_HOME/bin 目录中，并使用以下语法：

```
EAP_HOME/bin/wsconsume.sh [options] <wsdl-url>
```



注意

将 wsconsume.bat 脚本用于 Windows。

示例用法：

- **从 Example.wsdl WSDL 文件中生成 Java 类文件：**

```
EAP_HOME/bin/wsconsume.sh Example.wsdl
```

- **从 Example.wsdl WSDL 文件中生成 Java 源和类文件：**

```
EAP_HOME/bin/wsconsume.sh -k Example.wsdl
```

- **在 Example.wsdl WSDL 文件中的 my.org 软件包中生成 Java 源和类文件：**

```
EAP_HOME/bin/wsconsume.sh -k -p my.org Example.wsdl
```

- **使用多个绑定文件生成 Java 源和类文件：**

```
EAP_HOME/bin/wsconsume.sh -k -b schema-binding1.xsd -b schema-binding2.xsd
Example.wsdl
```

- **在 *JAVA_OPTS* 环境变量中配置 -Dlog4j 属性，以用于 wsconsume.sh 脚本：**
    - **此配置允许您创建可自定义的日志记录文件并设置应用程序的日志级别**

此配置允许您创建可自定义的日志记录文件并设置应用程序的日志级别。

- 要载入日志记录配置文件，请使用 **-Dlog4j.configuration=file:log4j.prsoperties** 或 **-Dlog4j.configuration=file:log4j.xml**，如下例所示：

```
JAVA_OPTS="-Dlog4j.configuration=file:log4j.properties" ./bin/wsconsume.sh
<WSDL_URL>
```

  其中，< ***WSDL_URL*** > 代表生成代码时要使用的 WSDL 文件路径。

使用 --help 参数或查看下表中所有可用 ws Consume 选项 的列表。

表 A.6. wssume 选项

选项	描述
-a, --additionalHeaders	启用对隐式 SOAP 标头的处理。
-b, --binding=<file>	一个或多个 Jakarta XML Web Services 或 Jakarta XML Binding 绑定文件。
-c --catalog=<file>	用于实体解析的 Oasis XML 目录文件.
-d --encoding=<charset>	用于生成的源的 charset 编码。
-e, --extension	启用 SOAP 1.2 绑定扩展。
-h, --help	显示此帮助消息.
-j --clientjar=<name>	为生成的构件创建一个 JAR 文件，以调用 Web 服务。
-k, --keep	保留/生成 Java 源.
-l, --load-consumer	加载使用者并退出（调试实用程序）。
-n, --nocompile	不要编译生成的源。
-o, --output=<directory>	用于放置生成的工件的目录。
-p --package=<name>	生成源的目标软件包。
-q、--quiet	有点静音。
-s, --source=<directory>	用于放置 Java 源的目录。

选项	描述
-t, --target=<2.1|2.2>	Jakarta XML Web 服务规范目标.
-v, --verbose	显示完整的异常堆栈跟踪。
-w --wsdlLocation=<loc>	值，用于 **@WebService.wsdlLocation**。

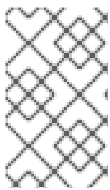
wsprovide

wsprovide 是随 JBoss EAP 提供的命令行工具，可为服务端点实施生成可移植 Jakarta XML Web 服务构件。它还具有可生成 WSDL 文件的选项。

使用

wsprovide 工具位于 **EAP_HOME/bin** 目录中，使用下列语法：

```
EAP_HOME/bin/wsprovide.sh [options] <endpoint class name>
```



注意

将 **wsprovide.bat** 脚本用于 Windows。

示例用法：

- 在 输出 目录中为可移植工件生成打包程序类。

```
EAP_HOME/bin/wsprovide.sh -o output my.package.MyEndpoint
```

- 在 输出 目录中生成 wrapper 类和 WSDL。

```
EAP_HOME/bin/wsprovide.sh -o output -w my.package.MyEndpoint
```

- 在 输出 目录中为引用其他 JAR 的端点生成 wrapper 类。

```
EAP_HOME/bin/wsprovide.sh -o output -c myapplication1.jar:myapplication2.jar
my.org.MyEndpoint
```

- 在 *JAVA_OPTS* 环境变量中配置 -Dlog4j 属性，以用于 wsprovide.sh 脚本：
  - 此配置允许您创建可自定义的日志记录文件并设置应用程序的日志级别。
  - 要载入日志记录配置文件，请使用 -Dlog4j.configuration=file:log4j.properties 或 -Dlog4j.configuration=file:log4j.xml，如下例所示：

```
JAVA_OPTS="-Dlog4j.configuration=file:log4j.properties" ./bin/wsprovide.sh
```

使用 --help 参数或查看下表中所有可用 wsprovide 选项的列表。

表 A.7. wsprovide 选项

选项	描述
-a, --address=<address>	WSDL 中生成的端口 soap:address。
-c, --classpath=<path>	包含端点的类路径。
-e, --extension	启用 SOAP 1.2 绑定扩展。
-h, --help	显示此帮助消息.
-k, --keep	保留/生成 Java 源.
-l, --load-provider	加载提供程序并退出（调试实用程序）。
-o, --output=<directory>	用于放置生成的工件的目录。
-q、--quiet	有点静音。
-r, --resource=<directory>	用于放置资源工件的目录。
-s, --source=<directory>	用于放置 Java 源的目录。
-t, --show-traces	显示完整的异常堆栈跟踪。
-w, --wsdl	启用 WSDL 文件生成.

A.7. JAKARTA XML WEB 服务通用 API 参考

Web 服务端点和客户端之间共享几个 Jakarta XML Web 服务开发概念。其中包括处理程序框架、消息上下文和错误处理。

处理程序框架

处理程序框架通过在客户端运行时和端点（服务器组件）绑定的 Jakarta XML Web Services 协议来实施。代理和 Dispatch 实例统称为绑定供应商，每个实例都使用协议绑定将其抽象功能绑定到特定的协议。

客户端和服务器端处理程序组织成一个有序的列表，称为处理程序链。每次发送或收到消息时，都会调用处理程序链中的处理程序。入站消息由处理程序处理，然后绑定提供程序处理它们。出站消息在绑定提供程序处理后由处理程序处理。

通过消息上下文调用处理程序，提供访问和修改入站和出站消息的方法，以及管理一组属性。消息上下文属性有助于各个处理程序之间的通信，以及处理程序与客户端和服务实施之间的通信。不同类型的处理程序通过不同类型的消息上下文调用。

逻辑处理程序

逻辑处理程序仅对消息上下文属性和消息有效负载执行操作。逻辑处理程序是独立于协议的，不能影响消息的特定协议部分。逻辑处理程序实施接口 javax.xml.ws.handler.LogicalHandler.

协议处理程序

协议处理程序对消息上下文属性和特定于协议的消息执行操作。协议处理程序特定于特定的协议，可以访问和更改消息的特定协议方面。协议处理程序实施派生自 javax.xml.ws.handler.Handler 的任何接口，但 javax.xml.ws.handler.LogicalHandler.

服务端点处理程序

在服务端点上，处理程序利用 @HandlerChain 注释来定义。处理程序链文件的位置可以是 externalForm 中的绝对 java.net.URL，也可以是来自源文件或类文件的相对路径。

```
@WebService
@HandlerChain(file = "jaxws-server-source-handlers.xml")
public class SOAPEndpointSourceImpl {
   ...
}
```

服务客户端处理程序

在 Jakarta XML Web Services 客户端上，处理程序通过使用 @HandlerChain 注释（如服务端点）或使用 Jakarta XML Web Services API 动态定义。

```
Service service = Service.create(wsdlURL, serviceName);
```

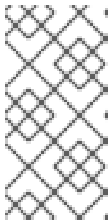
```
Endpoint port = (Endpoint)service.getPort(Endpoint.class);

BindingProvider bindingProvider = (BindingProvider)port;
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(new LogHandler());
handlerChain.add(new AuthorizationHandler());
handlerChain.add(new RoutingHandler());
bindingProvider.getBinding().setHandlerChain(handlerChain);
```

需要调用 setHandlerChain 方法。

消息上下文

MessageContext 界面是所有 Jakarta XML Web Services 消息上下文的超级界面。它通过 额外的方法和常量扩展 Map<String,Object> 来管理一组属性，使处理程序链中的处理程序能够共享相关状态。例如，处理程序可能使用 put 方法将 属性插入到消息上下文中。之后，处理程序链中的一个或多个处理程序可使用 get 方法获取消息。

属性作为 APPLICATION 或 HANDLER 限定。所有属性都可用于特定端点的消息交换模式(MEP)实例的所有处理程序。例如，如果逻辑处理程序将 属性放入消息上下文中，该属性也可用于 MEP 实例执行期间链中的任何协议处理程序。



注意

异步消息交换模式(MEP)允许在 HTTP 连接级别异步发送和接收消息。您可以通过在请求上下文中设置其他属性来启用它。

作用于 APPLICATION 级别的属性也可用于客户端应用和服务端点实施。属性 的默认作用域 是 HANDLER。

逻辑和 SOAP 消息使用不同的上下文。

逻辑消息上下文

调用逻辑处理程序时，它们会收到类型为 LogicalMessageContext 的消息上下文。LogicalMessageContext 使用获取和修改消息有效负载的方法扩展 MessageContext。它不提供对消息特定协议方面的访问权限。协议绑定定义通过逻辑消息上下文可以获得消息的哪些组件。SOAP 绑定中部署的逻辑处理器可以访问 SOAP 正文的内容，但不能访问 SOAP 标头。另一方面，XML/HTTP 绑定定义逻辑处理程序可以访问消息的整个 XML 有效负载。

SOAP 消息上下文

调用 SOAP 处理程序时，它们将接收 SOAPMessageContext。SOAPMessageContext 使用获取和修改 SOAP 消息有效负载的方法扩展 MessageContext。

错误处理

应用程序可能会引发 SOAPFaultException 或特定于应用程序的用户异常。对于后者，如果它们还不是部署的一部分，则在运行时生成所需的故障打包程序 Bean。

```
public void throwSoapFaultException() {
  SOAPFactory factory = SOAPFactory.newInstance();
  SOAPFault fault = factory.createFault("this is a fault string!", new QName("http://foo",
"FooCode"));
  fault.setFaultActor("mr.actor");
  fault.addDetail().addChildElement("test");
  throw new SOAPFaultException(fault);
}
```

```
public void throwApplicationException() throws UserException {
  throw new UserException("validation", 123, "Some validation error");
}
```

Jakarta XML Web 服务注释

Jakarta XML Web Services API 提供的注释在 Jakarta XML Web Services Specification 2.3 规范 2.3 规范 中定义，这些注释位于 javax.xml.ws 软件包中。

JWS API 提供的注解在 Web Services Metadata 中定义，遵循 Jakarta Web Services Metadata 规范 2.1 规范。这些注释位于 javax.jws 软件包中。

A.8. 高级 WS-TRUST 场景

A.8.1. 场景：SAML Holder-Of-Key Assertion 方案

WS-Trust 帮助管理软件安全令牌。SAML 断言是安全令牌的类型。在 Holder-Of-Key 方法中，STS 创建一个包含客户端公钥的 SAML 令牌，并使用其私钥为 SAML 令牌签名。客户端包括 SAML 令牌，并使用其私钥将传出的 soap 信封签名给 web 服务。Web 服务验证 SOAP 消息和 SAML 令牌。

实施此场景需要以下内容：

- 必须保护带有 Holder-Of-Key 主题确认方法的 SAML 令牌，以便无法侦听令牌。在大多数情况下，Holder-Of-Key 令牌与 HTTPS 相结合足以防止获取令牌。这意味着安全策略使用 sp:TransportBinding 和 sp:HttpsToken。

- Holder-Of-Key 令牌没有与其关联的加密或签名密钥，因此应该将 sp:IssuedToken of SymmetricKey 或 PublicKey keyType 用于 sp:SignedEndorsingSupportingTokens。

A.8.1.1. Web 服务提供商

本节列出了 SAML Holder-Of-Key 场景的 Web 服务元素。这些组件包括：

- Web 服务提供商 WSDL
- SSL 配置
- Web 服务提供商接口
- Web 服务提供商实施
- 加密属性和密钥存储文件
- 默认 MANIFEST.MF

A.8.1.1.1. Web 服务提供商 WSDL

Web 服务提供商是一个合同第一端点。HolderOfKeyService.wsdl WSDL 中声明的所有 WS-trust 和安全策略。在这种情况下，需要一个 ws-requester 来提供 SAML 2.0 令牌的 SymmetricKey keyType（由指定的 STS 发布）。STS 地址在 WSDL 中提供。使用传输绑定策略。该令牌被声明为签名和签名，SignedEndorsingSupportingTokens。

以下列表中的注释中提供了安全设置的详细解释：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-
extensions/holderofkeywssecuritypolicy"
             name="HolderOfKeyService"
        xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/holderofkeywssecuritypolicy"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```
        xmlns:wsp="http://www.w3.org/ns/ws-policy"
        xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsaws="http://www.w3.org/2005/08/addressing"
    xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-
extensions/holderofkeywssecuritypolicy"
                schemaLocation="HolderOfKeyService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="HolderOfKeyIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
<!--
        The wsp:PolicyReference binds the security requirements on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2HolderOfKeyPolicy" element is defined later in
this file.
-->
  <binding name="HolderOfKeyServicePortBinding" type="tns:HolderOfKeyIface">
    <wsp:PolicyReference URI="#TransportSAML2HolderOfKeyPolicy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
<!--
  The soap:address has been defined to use JBoss's https port, 8443.  This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
  <service name="HolderOfKeyService">
    <port name="HolderOfKeyServicePort" binding="tns:HolderOfKeyServicePortBinding">
      <soap:address location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-
holderofkey/HolderOfKeyService"/>
    </port>
  </service>
```

```
<wsp:Policy wsu:Id="TransportSAML2HolderOfKeyPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
  <!--
      The wsam:Addressing element, indicates that the endpoints of this
      web service MUST conform to the WS-Addressing specification.  The
      attribute wsp:Optional="false" enforces this assertion.
  -->
      <wsam:Addressing wsp:Optional="false">
        <wsp:Policy />
      </wsam:Addressing>
<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https.  WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
-->
        <sp:TransportBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <sp:TransportToken>
              <wsp:Policy>
                <sp:HttpsToken>
                  <wsp:Policy/>
                </sp:HttpsToken>
              </wsp:Policy>
            </sp:TransportToken>
<!--
    The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
    be used in performing cryptographic operations.
-->
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:TripleDes />
              </wsp:Policy>
            </sp:AlgorithmSuite>
<!--
    The sp:Layout element,  indicates the layout rules to apply when adding
    items to the security header.  The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax />
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp />
          </wsp:Policy>
        </sp:TransportBinding>

<!--
  The sp:SignedEndorsingSupportingTokens, when transport level security level is
  used there will be no message signature and the signature generated by the
  supporting token will sign the Timestamp.
-->
```

```
      <sp:SignedEndorsingSupportingTokens
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
<!--
  The sp:IssuedToken element asserts that a SAML 2.0 security token of type
  Bearer is expected from the STS.  The
  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
  attribute instructs the runtime to include the initiator's public key
  with every message sent to the recipient.

  The sp:RequestSecurityTokenTemplate element directs that all of the
  children of this element will be copied directly into the body of the
  RequestSecurityToken (RST) message that is sent to the STS when the
  initiator asks the STS to issue a token.
-->
          <sp:IssuedToken
            sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
            <sp:RequestSecurityTokenTemplate>
              <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0</t:TokenType>
 <!--
   KeyType of "SymmetricKey", the client must prove to the WS service that it
   possesses a particular symmetric session key.
 -->
              <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-
trust/200512/SymmetricKey</t:KeyType>
            </sp:RequestSecurityTokenTemplate>
            <wsp:Policy>
              <sp:RequireInternalReference />
            </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
            <sp:Issuer>
              <wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-
trust-sts-holderofkey/SecurityTokenService</wsaws:Address>
              <wsaws:Metadata
                xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
                wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
sts-holderofkey/SecurityTokenService?wsdl">
                <wsaw:ServiceName
                  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
                  xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
                  EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
              </wsaws:Metadata>
            </sp:Issuer>

          </sp:IssuedToken>
        </wsp:Policy>
      </sp:SignedEndorsingSupportingTokens>
<!--
   The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
   to be supported by the STS.  These particular elements generally refer
```

```
    to how keys are referenced within the SOAP envelope.  These are normally handled by
Apache CXF.
-->
        <sp:Wss11>
          <wsp:Policy>
            <sp:MustSupportRefIssuerSerial />
            <sp:MustSupportRefThumbprint />
            <sp:MustSupportRefEncryptedKey />
          </wsp:Policy>
        </sp:Wss11>
<!--
    The sp:Trust13 element declares controls for WS-Trust 1.3 options.
    They are policy assertions related to exchanges specifically with
    client and server challenges and entropy behaviors.  Again these are
    normally handled by Apache CXF.
-->
        <sp:Trust13>
          <wsp:Policy>
            <sp:MustSupportIssuedTokens />
            <sp:RequireClientEntropy />
            <sp:RequireServerEntropy />
          </wsp:Policy>
        </sp:Trust13>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</definitions>
```

A.8.1.1.2. SSL 配置

此 Web 服务使用 HTTPS，因此必须将 JBoss EAP 服务器配置为在 undertow 子系统中提供 SSL/TLS 支持。

有关如何为 Web 应用程序配置 HTTPS 的详情，请参考如何配置 *服务器安全性为应用程序配置单向和双向 SSL/TLS*。

A.8.1.1.3. Web 服务提供商接口

Web 服务提供商接口 HolderOfKeyIface 类是一个简单的 Web 服务定义。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-
```

```
extensions/holderofkeywssecuritypolicy"
)
public interface HolderOfKeyIface {
   @WebMethod
   String sayHello();
}
```

A.8.1.1.4. Web 服务提供商实施

Web 服务提供商实施 HolderOfKeyImpl 类是一个简单的 POJO。它使用标准的 WebService 注释来定义服务端点。此外，还有两个 Apache CXF 注解，Ed pointProperties 和 EndpointProperty 用于为 Apache CXF 运行时配置端点。这些注释来自 Apache WSS4J 项目，该项目为 Web 服务提供了主 WS-Security 标准的 Java 实施。这些注释以编程方式向端点添加属性。使用普通 Apache CXF 时，这些属性通常使用 Spring 配置中的 <jaxws:properties> 元素进行设置。这些注释允许在代码中配置属性。

WSS4J 使用 Crypto 接口获取用于签名创建/验证的密钥和证书，如 WSDL 为此服务的要求。HolderOfKeyImpl 提供的 WSS4J 配置信息用于 Crypto 的 Merlin 实施。

列表中的第一个 EndpointProperty 语句禁用了对基本安全配置文件 1.1 的合规性。下一个 EndpointProperty 语句声明包含(Merlin)Crypto 配置信息的 Java 属性文件。最后的 EndpointProperty 语句声明 STSHolderOfKeyCallbackHandler 实施类。它用于获取密钥存储文件中证书的密码。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
   (
      portName = "HolderOfKeyServicePort",
      serviceName = "HolderOfKeyService",
      wsdlLocation = "WEB-INF/wsdl/HolderOfKeyService.wsdl",
      targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/holderofkeywssecuritypolicy",
      endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyIface"
   )
@EndpointProperties(value = {
   @EndpointProperty(key = "ws-security.is-bsp-compliant", value = "false"),
   @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties"),
   @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.holderofkey.HolderOfKeyCallbackHandler
")
})
public class HolderOfKeyImpl implements HolderOfKeyIface {
   public String sayHello() {
```

```
        return "Holder-Of-Key WS-Trust Hello World!";
    }
}
```

A.8.1.1.5. 加密属性和密钥存储文件

WSS4J 的 Crypto 实施通过包含 Crypto 配置数据的 Java 属性文件来加载和配置。文件包含特定于实施的属性，如密钥存储位置、密码、默认别名等。这个应用程序使用 Merlin 实施。serviceKeystore.properties 文件包含此信息。

servicestore.jks 文件是 Java KeyStore(JKS)存储库。它包含 myservicekey 和 mystskey 的自签名证书。

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

A.8.1.1.6. 默认 MANIFEST.MF

此应用程序需要访问 org.jboss.ws.cxf.jbossws-cxf 中提供的 JBossWS 和 Apache CXF API。dependency 语句指示服务器在部署时提供它们。

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

A.8.2. 场景：SAML Bearerions

WS-Trust 管理软件安全令牌。SAML 断言是安全令牌的类型。在 SAML Bearer 场景中，服务提供商会在服务验证令牌签名后自动信任传入的 SOAP 请求来自 SAML 令牌中定义的主题。

实施此方案需要满足以下要求：

- 带有 Bearer 主题确认方法的 SAML 令牌必须受到保护，以便令牌不能被嗅探。在大多数情况下，bearer 令牌与 HTTPS 相结合足以防止"中间人"获得令牌。这意味着使用 sp: TransportBinding 和 sp: HttpsToken 的安全策略。

- bearer 令牌没有与之关联的加密或签名密钥，因此 bearer keyType 的 sp:IssuedToken 应当用于 sp:SupportingToken 或 sp:SignedSupportingTokens。

A.8.2.1. Web 服务提供商

本节探讨 SAML Bearer 场景的 Web 服务元素。这些组件包括：

- bearer Web Service Provider WSDL
- SSL 配置
- bearer Web 服务提供商接口
- bearer Web 服务提供商实施
- 加密属性和密钥存储文件
- 默认 MANIFEST.MF

A.8.2.1.1. bearer Web Service Provider WSDL

Web 服务提供商是一个合同第一端点。WS-信任和安全策略在 BearerService.wsdl WSDL 中声明。在这种情况下，Wer-requester 需要提供由指定的 STS 发布的 SAML 2.0 Bearer 令牌。STS 的地址在 WSDL 中提供。HTTPS、传输绑定和 HttpsToken 策略用于保护在 ws-requester 和 ws- provider 之间发送的消息的 SOAP 正文。安全设置详细信息在以下列表中作为注释提供：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
             name="BearerService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
             xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">

  <types>
```

```
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
              schemaLocation="BearerService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="BearerIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>

<!--
        The wsp:PolicyReference binds the security requirments on all the endpoints.
        The wsp:Policy wsu:Id="#TransportSAML2BearerPolicy" element is defined later in this
file.
-->
  <binding name="BearerServicePortBinding" type="tns:BearerIface">
    <wsp:PolicyReference URI="#TransportSAML2BearerPolicy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

<!--
  The soap:address has been defined to use JBoss's https port, 8443.  This is
  set in conjunction with the sp:TransportBinding policy for https.
-->
  <service name="BearerService">
    <port name="BearerServicePort" binding="tns:BearerServicePortBinding">
      <soap:address location="https://@jboss.bind.address@:8443/jaxws-samples-wsse-policy-trust-
bearer/BearerService"/>
    </port>
  </service>


  <wsp:Policy wsu:Id="TransportSAML2BearerPolicy">
    <wsp:ExactlyOne>
      <wsp:All>
  <!--
        The wsam:Addressing element, indicates that the endpoints of this
        web service MUST conform to the WS-Addressing specification.  The
        attribute wsp:Optional="false" enforces this assertion.
```

```
  -->
      <wsam:Addressing wsp:Optional="false">
        <wsp:Policy />
      </wsam:Addressing>

<!--
  The sp:TransportBinding element indicates that security is provided by the
  message exchange transport medium, https.  WS-Security policy specification
  defines the sp:HttpsToken for use in exchanging messages transmitted over HTTPS.
-->
      <sp:TransportBinding
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
          <sp:TransportToken>
            <wsp:Policy>
              <sp:HttpsToken>
                <wsp:Policy/>
              </sp:HttpsToken>
            </wsp:Policy>
          </sp:TransportToken>
<!--
    The sp:AlgorithmSuite element, requires the TripleDes algorithm suite
    be used in performing cryptographic operations.
-->
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:TripleDes />
            </wsp:Policy>
          </sp:AlgorithmSuite>
<!--
    The sp:Layout element,  indicates the layout rules to apply when adding
    items to the security header.  The sp:Lax sub-element indicates items
    are added to the security header in any order that conforms to
    WSS: SOAP Message Security.
-->
          <sp:Layout>
            <wsp:Policy>
              <sp:Lax />
            </wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp />
        </wsp:Policy>
      </sp:TransportBinding>

<!--
  The sp:SignedSupportingTokens element causes the supporting tokens
  to be signed using the primary token that is used to sign the message.
-->
      <sp:SignedSupportingTokens
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        <wsp:Policy>
<!--
  The sp:IssuedToken element asserts that a SAML 2.0 security token of type
  Bearer is expected from the STS.  The
  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
```

```
  attribute instructs the runtime to include the initiator's public key
  with every message sent to the recipient.

  The sp:RequestSecurityTokenTemplate element directs that all of the
  children of this element will be copied directly into the body of the
  RequestSecurityToken (RST) message that is sent to the STS when the
  initiator asks the STS to issue a token.
-->
          <sp:IssuedToken
           sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
           <sp:RequestSecurityTokenTemplate>
            <t:TokenType>http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-
1.1#SAMLV2.0</t:TokenType>
            <t:KeyType>http://docs.oasis-open.org/ws-sx/ws-trust/200512/Bearer</t:KeyType>
           </sp:RequestSecurityTokenTemplate>
           <wsp:Policy>
            <sp:RequireInternalReference />
           </wsp:Policy>
<!--
  The sp:Issuer element defines the STS's address and endpoint information
  This information is used by the STSClient.
-->
           <sp:Issuer>
            <wsaws:Address>http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-
trust-sts-bearer/SecurityTokenService</wsaws:Address>
            <wsaws:Metadata
             xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
             wsdli:wsdlLocation="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
sts-bearer/SecurityTokenService?wsdl">
             <wsaw:ServiceName
              xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
              xmlns:stsns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
              EndpointName="UT_Port">stsns:SecurityTokenService</wsaw:ServiceName>
            </wsaws:Metadata>
           </sp:Issuer>

          </sp:IssuedToken>
        </wsp:Policy>
       </sp:SignedSupportingTokens>
<!--
   The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
   to be supported by the STS.  These particular elements generally refer
   to how keys are referenced within the SOAP envelope.  These are normally handled by
Apache CXF.
-->
       <sp:Wss11>
        <wsp:Policy>
          <sp:MustSupportRefIssuerSerial />
          <sp:MustSupportRefThumbprint />
          <sp:MustSupportRefEncryptedKey />
        </wsp:Policy>
       </sp:Wss11>
<!--
   The sp:Trust13 element declares controls for WS-Trust 1.3 options.
   They are policy assertions related to exchanges specifically with
```

```
    client and server challenges and entropy behaviors.  Again these are
    normally handled by Apache CXF.
-->
        <sp:Trust13>
          <wsp:Policy>
            <sp:MustSupportIssuedTokens />
            <sp:RequireClientEntropy />
            <sp:RequireServerEntropy />
          </wsp:Policy>
        </sp:Trust13>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</definitions>
```

A.8.2.1.2. SSL 配置

此 Web 服务使用 HTTPS，因此必须将 JBoss EAP 服务器配置为在 undertow 子系统中提供 SSL 支持。

有关如何为 Web 应用程序配置 HTTPS 的详情，请参考如何配置 *服务器安全性为应用程序配置单向和双向 SSL/TLS*。

A.8.2.1.3. bearer Web 服务提供商接口

BearerIface Bearer Web 服务提供商接口类是一个简单的 Web 服务定义。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy"
)
public interface BearerIface {
  @WebMethod
  String sayHello();
}
```

A.8.2.1.4. bearer Web Service Providers 实施

BearerImpl Web 服务提供商实施类是一个简单的 POJO。它使用标准的 WebService 注释来定义服务端点。此外，还有两个 Apache CXF 注解，Ed pointProperties 和 EndpointProperty 用于为 Apache CXF 运行时配置端点。这些注释来自 Apache WSS4J 项目，该项目为 Web 服务提供了主 WS-

Security 标准的 Java 实施。这些注释以编程方式向端点添加属性。使用普通 Apache CXF 时，这些属性通常使用 Spring 配置中的 <jaxws:properties> 元素进行设置。这些注释允许在代码中配置属性。

WSS4J 使用 Crypto 接口获取用于签名创建/验证的密钥和证书，如 WSDL 为此服务的要求。BearerImpl 提供的 WSS4J 配置信息用于 Crypto 的 Merlin 实施。

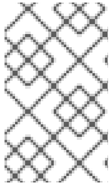
由于 Web 服务提供商自动信任来自 SAML 令牌中定义的主题的传入 SOAP 请求，所以与前面的示例中不同，Crypto CallbackHandler 类或签名用户名不需要该请求。但是，为了验证消息签名，仍然需要包含(Merlin)Crypto 配置信息的 Java 属性文件。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;

import javax.jws.WebService;

@WebService
(
  portName = "BearerServicePort",
  serviceName = "BearerService",
  wsdlLocation = "WEB-INF/wsdl/BearerService.wsdl",
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/bearerwssecuritypolicy",
  endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.trust.bearer.BearerIface"
)
@EndpointProperties(value = {
  @EndpointProperty(key = "ws-security.signature.properties", value =
"serviceKeystore.properties")
})
public class BearerImpl implements BearerIface {
  public String sayHello() {
    return "Bearer WS-Trust Hello World!";
  }
}
```

A.8.2.1.5. 加密属性和密钥存储文件

WSS4J 的 Crypto 实施通过包含 Crypto 配置数据的 Java 属性文件来加载和配置。文件包含特定于实施的属性，如密钥存储位置、密码、默认别名等。此应用程序正在使用 Merlin 实施。serviceKeystore.properties 文件包含此信息。

servicestore.jks 文件是 Java KeyStore(JKS)存储库。它包含 myservicekey 和 mystskey 的自签名证书。



注意

自签名证书不适合在生产环境中使用。

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=sspass
org.apache.ws.security.crypto.merlin.keystore.alias=myservicekey
org.apache.ws.security.crypto.merlin.keystore.file=servicestore.jks
```

A.8.2.1.6. 默认 MANIFEST.MF

部署时，此应用程序需要访问模块 org.jboss.ws.cxf.jbossws-cxf.jbossws-cxf-client 提供的 JBossWS 和 Apache CXF API。dependency 语句指示服务器在部署时提供它们。

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client
```

A.8.2.2. bearer 安全令牌服务

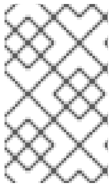
本节列出了提供 SAML Bearer 令牌提供安全令牌服务功能的关键元素。这些组件包括：

- 安全域
- STS WSDL
- STS 实施类
- STSBearerCallbackHandler Class
- 加密属性和密钥存储文件
- 默认 MANIFEST.MF

A.8.2.2.1. 安全域

STS 需要配置 JBoss 安全域。jboss-web.xml 描述符声明了已命名的安全域JBossWS-trust-st，供此服务用于身份验证。此安全域需要两个属性文件，并在 JBoss EAP 服务器配置文件中添加安全域声明。

在这种情况下，域需要包含用户 alice、密码 clarinet 和角色 朋友。请参考以下关于 jbossws-users.properties 和 jbossws-roles.properties 的列表 ：此外，必须将以下 XML 添加到服务器配置文件中的 JBoss security 子系统中：



注意

将"SOME_PATH"替换为适当的信息。

```
<security-domain name="JBossWS-trust-sts">
  <authentication>
    <login-module code="UsersRoles" flag="required">
      <module-option name="usersProperties" value="/SOME_PATH/jbossws-users.properties"/>
      <module-option name="unauthenticatedIdentity" value="anonymous"/>
      <module-option name="rolesProperties" value="/SOME_PATH/jbossws-roles.properties"/>
    </login-module>
  </authentication>
</security-domain>
```

示例： jboss-web.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss-web PUBLIC "-//JBoss//DTD Web Application 2.4//EN" ">
<jboss-web>
  <security-domain>java:/jaas/JBossWS-trust-sts</security-domain>
</jboss-web>
```

示例： jbossws-users.properties 文件

```
# A sample users.properties file for use with the UsersRolesLoginModule
alice=clarinet
```

示例： jbossws-roles.properties 文件

```
# A sample roles.properties file for use with the UsersRolesLoginModule
alice=friend
```

A.8.2.2.2. STS WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:tns="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wstrust="http://docs.oasis-open.org/ws-sx/ws-trust/200512/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsap10="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">

  <wsdl:types>
    <xs:schema elementFormDefault="qualified"
            targetNamespace='http://docs.oasis-open.org/ws-sx/ws-trust/200512'>

      <xs:element name='RequestSecurityToken'
            type='wst:AbstractRequestSecurityTokenType'/>
      <xs:element name='RequestSecurityTokenResponse'
            type='wst:AbstractRequestSecurityTokenType'/>

      <xs:complexType name='AbstractRequestSecurityTokenType'>
        <xs:sequence>
          <xs:any namespace='##any' processContents='lax' minOccurs='0'
            maxOccurs='unbounded'/>
        </xs:sequence>
        <xs:attribute name='Context' type='xs:anyURI' use='optional'/>
        <xs:anyAttribute namespace='##other' processContents='lax'/>
      </xs:complexType>
      <xs:element name='RequestSecurityTokenCollection'
            type='wst:RequestSecurityTokenCollectionType'/>
      <xs:complexType name='RequestSecurityTokenCollectionType'>
        <xs:sequence>
          <xs:element name='RequestSecurityToken'
              type='wst:AbstractRequestSecurityTokenType' minOccurs='2'
              maxOccurs='unbounded'/>
        </xs:sequence>
      </xs:complexType>
```

```
    <xs:element name='RequestSecurityTokenResponseCollection'
            type='wst:RequestSecurityTokenResponseCollectionType'/>
    <xs:complexType name='RequestSecurityTokenResponseCollectionType'>
      <xs:sequence>
        <xs:element ref='wst:RequestSecurityTokenResponse' minOccurs='1'
              maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:anyAttribute namespace='##other' processContents='lax'/>
    </xs:complexType>

  </xs:schema>
</wsdl:types>

<!-- WS-Trust defines the following GEDs -->
<wsdl:message name="RequestSecurityTokenMsg">
  <wsdl:part name="request" element="wst:RequestSecurityToken"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseMsg">
  <wsdl:part name="response"
          element="wst:RequestSecurityTokenResponse"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenCollectionMsg">
  <wsdl:part name="requestCollection"
          element="wst:RequestSecurityTokenCollection"/>
</wsdl:message>
<wsdl:message name="RequestSecurityTokenResponseCollectionMsg">
  <wsdl:part name="responseCollection"
          element="wst:RequestSecurityTokenResponseCollection"/>
</wsdl:message>

<!-- This portType an example of a Requestor (or other) endpoint that
Accepts SOAP-based challenges from a Security Token Service -->
<wsdl:portType name="WSSecurityRequestor">
  <wsdl:operation name="Challenge">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an STS supporting full protocol -->
<!--
    The wsdl:portType and data types are XML elements defined by the
    WS_Trust specification.  The wsdl:portType defines the endpoints
    supported in the STS implementation.  This WSDL defines all operations
    that an STS implementation can support.
-->
<wsdl:portType name="STS">
  <wsdl:operation name="Cancel">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/CancelFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Issue">
```

```
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTRC/IssueFinal"
      message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/RenewFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/ValidateFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <wsdl:input
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KET"
      message="tns:RequestSecurityTokenMsg"/>
    <wsdl:output
      wsam:Action="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/KETFinal"
      message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
    <wsdl:input message="tns:RequestSecurityTokenCollectionMsg"/>
    <wsdl:output message="tns:RequestSecurityTokenResponseCollectionMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!-- This portType is an example of an endpoint that accepts
Unsolicited RequestSecurityTokenResponse messages -->
<wsdl:portType name="SecurityTokenResponseService">
  <wsdl:operation name="RequestSecurityTokenResponse">
    <wsdl:input message="tns:RequestSecurityTokenResponseMsg"/>
  </wsdl:operation>
</wsdl:portType>

<!--
   The wsp:PolicyReference binds the security requirments on all the STS endpoints.
   The wsp:Policy wsu:Id="UT_policy" element is later in this file.
-->
<wsdl:binding name="UT_Binding" type="wstrust:STS">
  <wsp:PolicyReference URI="#UT_policy"/>
  <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="Issue">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Issue"/>
```

```
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Validate">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Validate"/>
    <wsdl:input>
      <wsp:PolicyReference
        URI="#Input_policy"/>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <wsp:PolicyReference
        URI="#Output_policy"/>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Cancel">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Cancel"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="Renew">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/Renew"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="KeyExchangeToken">
    <soap:operation
      soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/KeyExchangeToken"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="RequestCollection">
```

```
      <soap:operation
        soapAction="http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/RequestCollection"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="SecurityTokenService">
    <wsdl:port name="UT_Port" binding="tns:UT_Binding">
      <soap:address location="http://localhost:8080/SecurityTokenService/UT"/>
    </wsdl:port>
  </wsdl:service>


  <wsp:Policy wsu:Id="UT_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <!--
          The sp:UsingAddressing element, indicates that the endpoints of this
          web service conforms to the WS-Addressing specification.  More detail
          can be found here: [http://www.w3.org/TR/2006/CR-ws-addr-wsdl-20060529]
        -->
        <wsap10:UsingAddressing/>
        <!--
          The sp:SymmetricBinding element indicates that security is provided
          at the SOAP layer and any initiator must authenticate itself by providing
          WSS UsernameToken credentials.
        -->
        <sp:SymmetricBinding
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <wsp:Policy>
            <!--
              In a symmetric binding, the keys used for encrypting and signing in both
              directions are derived from a single key, the one specified by the
              sp:ProtectionToken element.  The sp:X509Token sub-element declares this
              key to be a X.509 certificate and the
              IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never"
              attribute adds the requirement that the token MUST NOT be included in
              any messages sent between the initiator and the recipient; rather, an
              external reference to the token should be used.  Lastly the WssX509V3Token10
              sub-element declares that the Username token presented by the initiator
              should be compliant with Web Services Security UsernameToken Profile
              1.0 specification. [ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
username-token-profile-1.0.pdf ]
            -->
            <sp:ProtectionToken>
              <wsp:Policy>
                <sp:X509Token
                  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never">
                  <wsp:Policy>
```

```
            <sp:RequireDerivedKeys/>
            <sp:RequireThumbprintReference/>
            <sp:WssX509V3Token10/>
          </wsp:Policy>
        </sp:X509Token>
      </wsp:Policy>
    </sp:ProtectionToken>
    <!--
      The sp:AlgorithmSuite element, requires the Basic256 algorithm suite
      be used in performing cryptographic operations.
    -->
    <sp:AlgorithmSuite>
      <wsp:Policy>
        <sp:Basic256/>
      </wsp:Policy>
    </sp:AlgorithmSuite>
    <!--
      The sp:Layout element,  indicates the layout rules to apply when adding
      items to the security header.  The sp:Lax sub-element indicates items
      are added to the security header in any order that conforms to
      WSS: SOAP Message Security.
    -->
    <sp:Layout>
      <wsp:Policy>
        <sp:Lax/>
      </wsp:Policy>
    </sp:Layout>
    <sp:IncludeTimestamp/>
    <sp:EncryptSignature/>
    <sp:OnlySignEntireHeadersAndBody/>
  </wsp:Policy>
</sp:SymmetricBinding>

<!--
  The sp:SignedSupportingTokens element declares that the security header
  of messages must contain a sp:UsernameToken and the token must be signed.
  The attribute IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient"
  on sp:UsernameToken indicates that the token MUST be included in all
  messages sent from initiator to the recipient and that the token MUST
  NOT be included in messages sent from the recipient to the initiator.
  And finally the element sp:WssUsernameToken10 is a policy assertion
  indicating the Username token should be as defined in  Web Services
  Security UsernameToken Profile 1.0
-->
<sp:SignedSupportingTokens
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:Policy>
    <sp:UsernameToken
      sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
      <wsp:Policy>
        <sp:WssUsernameToken10/>
      </wsp:Policy>
    </sp:UsernameToken>
  </wsp:Policy>
```

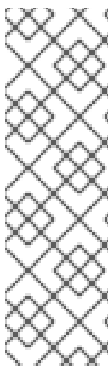
```
        </sp:SignedSupportingTokens>
        <!--
          The sp:Wss11 element declares WSS: SOAP Message Security 1.1 options
          to be supported by the STS.  These particular elements generally refer
          to how keys are referenced within the SOAP envelope.  These are normally
          handled by Apache CXF.
        -->
        <sp:Wss11
         xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <wsp:Policy>
          <sp:MustSupportRefKeyIdentifier/>
          <sp:MustSupportRefIssuerSerial/>
          <sp:MustSupportRefThumbprint/>
          <sp:MustSupportRefEncryptedKey/>
         </wsp:Policy>
        </sp:Wss11>
        <!--
          The sp:Trust13 element declares controls for WS-Trust 1.3 options.
          They are policy assertions related to exchanges specifically with
          client and server challenges and entropy behaviors.  Again these are
          normally handled by Apache CXF.
        -->
        <sp:Trust13
         xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <wsp:Policy>
          <sp:MustSupportIssuedTokens/>
          <sp:RequireClientEntropy/>
          <sp:RequireServerEntropy/>
         </wsp:Policy>
        </sp:Trust13>
       </wsp:All>
      </wsp:ExactlyOne>
     </wsp:Policy>

     <wsp:Policy wsu:Id="Input_policy">
      <wsp:ExactlyOne>
       <wsp:All>
        <sp:SignedParts
         xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
         <sp:Body/>
         <sp:Header Name="To"
               Namespace="http://www.w3.org/2005/08/addressing"/>
         <sp:Header Name="From"
               Namespace="http://www.w3.org/2005/08/addressing"/>
         <sp:Header Name="FaultTo"
               Namespace="http://www.w3.org/2005/08/addressing"/>
         <sp:Header Name="ReplyTo"
               Namespace="http://www.w3.org/2005/08/addressing"/>
         <sp:Header Name="MessageID"
               Namespace="http://www.w3.org/2005/08/addressing"/>
         <sp:Header Name="RelatesTo"
               Namespace="http://www.w3.org/2005/08/addressing"/>
         <sp:Header Name="Action"
               Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
       </wsp:All>
```

```
    </wsp:ExactlyOne>
  </wsp:Policy>

  <wsp:Policy wsu:Id="Output_policy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:SignedParts
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
          <sp:Body/>
          <sp:Header Name="To"
                Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="From"
                Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="FaultTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="ReplyTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="MessageID"
                Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="RelatesTo"
                Namespace="http://www.w3.org/2005/08/addressing"/>
          <sp:Header Name="Action"
                Namespace="http://www.w3.org/2005/08/addressing"/>
        </sp:SignedParts>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

</wsdl:definitions>
```

A.8.2.2.3. STS 实施类

Apache CXF 的 STS SecurityTokenServiceProvider 是符合 WS-Trust 规范中定义的协议和功能的 Web 服务提供商。它具有模块化架构，其组件可配置或可替换。实施和配置插件可以启用可选功能。您可以通过从 SecurityTokenServiceProvider 扩展并覆盖默认设置来自定义自己的 STS。

SampleSTSBearer STS 实施类是从 SecurityTokenServiceProvider 扩展的 POJO。



注意

SampleSTSBearer 类使用 WebServiceProvider 注释来定义，而不是使用 WebService 注释。此注释将服务定义为基于 提供程序的端点，它支持面向消息传递的 Web 服务方法。特别是，它表示交换的消息将是 XML 文档。SecurityTokenServiceProvider 是 javax.xml.ws.Provider 接口的一种实施。相比之下，WebService 注释定义基于服务端点接口的端点，该端点支持使用 SOAP 信封进行消息交换。

如 BearerImpl 类中所做的操作，WSS4J 注解 EndpointProperties 和 EndpointProperty 为

Apache CXF 运行时提供端点配置。列表中的第一个 EndpointProperty 语句声明了用于消息签名的用户名称。它在密钥存储中用作别名名称，用于获取用户签名的证书和私钥。接下来的两个 EndpointProperty 语句声明包含(Merlin)Crypto 配置信息的 Java 属性文件。在本例中，用于签名和加密消息。WSS4J 读取此文件，以及消息处理所需的信息。最后的 EndpointProperty 语句声明 STSBearerCallbackHandler 实施类。它用于获取密钥存储文件中证书的密码。

在这种实施中，我们自定义令牌规范、令牌验证及其静态属性的操作。

StaticSTSProperties 用于设置用于在 STS 中配置资源的选项。这看起来像是使用 WSS4J 注释进行的设置重复。值相同，但设置的下层结构不同，因此必须在两个位置声明此信息。

setIssuer 设置很重要，因为它唯一地标识了发行的 STS。签发者字符串嵌入了发布的令牌中，在验证令牌时，STS 会检查签发者字符串值。因此，务必要以一致的方式使用签发者字符串，以便 STS 可以识别发布的令牌。

setEndpoints 调用允许按地址声明一组允许的令牌接收者。地址指定为 reg-ex 模式。

TokenIssueOperation 具有一个模块化结构。这允许将自定义行为注入到消息处理中。在本例中，我们将覆盖 SecurityTokenServiceProvider 默认行为并执行 SAML 令牌处理。Apache CXF 提供了 SAMLTokenProvider 的一种实施，它可用于而不是创建 SAMLTokenProvider。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

import javax.xml.ws.WebServiceProvider;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/bearer-ws-trust-1.4-service.wsdl")
//dependency on org.apache.cxf module or on module that exports org.apache.cxf (e.g.
org.jboss.ws.cxf.jbossws-cxf-client) is needed, otherwise Apache CXF annotations are
ignored
@EndpointProperties(value = {
```

```
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer.STSBearerCallbackHandler")
})
public class SampleSTSBearer extends SecurityTokenServiceProvider {

  public SampleSTSBearer() throws Exception {
    super();

    StaticSTSProperties props = new StaticSTSProperties();
    props.setSignatureCryptoProperties("stsKeystore.properties");
    props.setSignatureUsername("mystskey");
    props.setCallbackHandlerClass(STSBearerCallbackHandler.class.getName());
    props.setEncryptionCryptoProperties("stsKeystore.properties");
    props.setEncryptionUsername("myservicekey");
    props.setIssuer("DoubleItSTSIssuer");

    List<ServiceMBean> services = new LinkedList<ServiceMBean>();
    StaticService service = new StaticService();
    service.setEndpoints(Arrays.asList(
      "https://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
      "https://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService",
      "https://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-bearer/BearerService"
    ));
    services.add(service);

    TokenIssueOperation issueOperation = new TokenIssueOperation();
    issueOperation.getTokenProviders().add(new SAMLTokenProvider());
    issueOperation.setServices(services);
    issueOperation.setStsProperties(props);
    this.setIssueOperation(issueOperation);
  }
}
```

A.8.2.2.4. STSBearerCallbackHandler Class

STSBearerCallbackHandler 是 WSS4J Crypto API 的回调处理程序。它用于在密钥存储中获取私钥的密码。此类使 Apache CXF 能够检索用于消息签名的用户名密码。

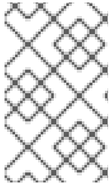
```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.stsbearer;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

import java.util.HashMap;
import java.util.Map;

public class STSBearerCallbackHandler extends PasswordCallbackHandler {
  public STSBearerCallbackHandler() {
    super(getInitMap());
  }
```

```
    private static Map<String, String> getInitMap() {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("mystskey", "stskpass");
      passwords.put("alice", "clarinet");
      return passwords;
    }
}
```

A.8.2.2.5. 加密属性和密钥存储文件

WSS4J 的 Crypto 实施通过包含 Crypto 配置数据的 Java 属性文件来加载和配置。文件包含特定于实施的属性，如密钥存储位置、密码、默认别名等。此应用程序正在使用 Merlin 实施。stsKeystore.properties 文件包含此信息。

servicestore.jks 文件是 Java KeyStore(JKS)存储库。它包含 myservicekey 和 mystskey 的自签名证书。



注意

自签名证书不适合在生产环境中使用。

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=stsspass
org.apache.ws.security.crypto.merlin.keystore.file=stsstore.jks
```

A.8.2.2.6. 默认 MANIFEST.MF

此应用程序需要访问 org.jboss.ws.cxf.jbossws-cxf 中提供的 JBossWS 和 Apache CXF API。还需要 org.jboss.ws.cxf.sts 模块在 SampleSTS 构造器中构建 STS 配置。dependency 语句指示服务器在部署时提供它们。

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client,org.jboss.ws.cxf.sts
```

A.8.2.3. Web Service Requester

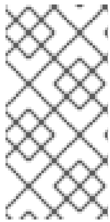
本节详细介绍了调用实现端点安全性的 Web 服务的关键元素，如 SAML Bearer 场景中所述。讨论的组件包括：

- Web 服务请求者实施

- ClientCallbackHandler
- 加密属性和密钥存储文件

A.8.2.3.1. Web 服务请求者实施

ws-requester （客户端）使用了标准步骤来创建对 Web 服务的引用。为解决端点安全要求，Web 服务的"请求上下文"配置了消息生成所需的信息。此外，与 STS 通信的 STSClient 也配置了类似的值。



注意

以 a .it 后缀结尾的密钥字符串将这些设置标记为属于 STSClient。内部 Apache CXF 代码将此信息分配到此服务调用自动生成的 STSClient。

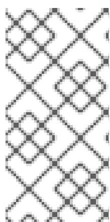
还有一种设置 STSCLient 的方法。用户可以提供自己的 STSClient 实例。Apache CXF 代码使用此对象，不自动生成一个对象。以这种方式提供 STSClient 时，用户必须为其提供 org.apache.cxf.Bus, 配置键不得具有 .it 后缀。这可用于 ActAs 和 OnBehalfOf 示例。

```
String serviceURL = "https://" + getServerHost() + ":8443/jaxws-samples-wsse-policy-trust-
bearer/BearerService";

final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/bearerwssecuritypolicy", "BearerService");
Service service = Service.create(new URL(serviceURL + "?wsdl"), serviceName);
BearerIface proxy = (BearerIface) service.getPort(BearerIface.class);

Map<String, Object> ctx = ((BindingProvider)proxy).getRequestContext();

// set the security related configuration information for the service "request"
ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
  Thread.currentThread().getContextClassLoader().getResource(
  "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");
ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

//-- Configuration settings that will be transfered to the STSClient
// "alice" is the name provided for the WSS Username. Her password will
// be retreived from the ClientCallbackHander by the STSClient.
ctx.put(SecurityConstants.USERNAME + ".it", "alice");
ctx.put(SecurityConstants.CALLBACK_HANDLER + ".it", new ClientCallbackHandler());
ctx.put(SecurityConstants.ENCRYPT_PROPERTIES + ".it",
```

```
    Thread.currentThread().getContextClassLoader().getResource(
    "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.ENCRYPT_USERNAME + ".it", "mystskey");
ctx.put(SecurityConstants.STS_TOKEN_USERNAME + ".it", "myclientkey");
ctx.put(SecurityConstants.STS_TOKEN_PROPERTIES + ".it",
    Thread.currentThread().getContextClassLoader().getResource(
    "META-INF/clientKeystore.properties"));
ctx.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO + ".it", "true");

proxy.sayHello();
```

A.8.2.3.2. ClientCallbackHandler

ClientCallbackHandler 是 WSS4J Crypto API 的回调处理程序。它用于在密钥存储中获取私钥的密码。此类使 Apache CXF 能够检索用于消息签名的用户名密码。



注意

这里提供了用户 alice 和密码。此信息不在(JKS)密钥存储中，而是在安全域中提供。它在 jbossws-users.properties 文件中声明。

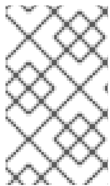
```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientCallbackHandler implements CallbackHandler {

  public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
     for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof WSPasswordCallback) {
           WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
           if ("myclientkey".equals(pc.getIdentifier())) {
              pc.setPassword("ckpass");
              break;
           } else if ("alice".equals(pc.getIdentifier())) {
              pc.setPassword("clarinet");
              break;
           } else if ("bob".equals(pc.getIdentifier())) {
              pc.setPassword("trombone");
              break;
           } else if ("myservicekey".equals(pc.getIdentifier())) {  // rls test  added for bearer test
             pc.setPassword("skpass");
             break;
           }
        }
```

```
        }
    }
}
```

A.8.2.3.3. 加密属性和密钥存储文件

WSS4J 的 Crypto 实施通过包含 Crypto 配置数据的 Java 属性文件来加载和配置。文件包含特定于实施的属性，如密钥存储位置、密码、默认别名等。此应用程序正在使用 Merlin 实施。clientKeystore.properties 文件包含此信息。

clientstore.jks 文件是 Java KeyStore(JKS)存储库。它包含 myservicekey 和 mystskey 的自签名证书。



注意

自签名证书不适合在生产环境中使用。

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=cspass
org.apache.ws.security.crypto.merlin.keystore.alias=myclientkey
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/clientstore.jks
```

A.8.3. 场景：OnehalfOf WS-Trust

OnBehalfOf 功能用于使用代理模式的情况。在这种情况下，客户端无法直接访问 STS，而是通过代理网关进行通信。代理网关验证调用者，并将调用者的信息放入 RequestSecurityToken (RST)的 OnBehalfOf 元素中，以用于处理。生成的令牌仅包含与代理客户端相关的声明，使代理对颁发的令牌的接收器完全透明。

OnBehalfOf 不仅仅是 RST 中的新子元素。它在与 STS 协商令牌时提供有关原始调用者的额外信息。OnBehalfOf 元素通常采用令牌形式，其身份声明（如名称、角色和授权代码）供客户端访问服务。

OnBehalfOf 方案是基本 WS-Trust 场景的扩展。在本例中，OnBehalfOf 服务代表用户调用 ws-service。对基本场景的代码仅作几处补充。添加了 OnBehalfOf Web 服务提供商和回调处理程序。OnBehalfOf Web 服务的 WSDL 采用了与 ws-provider 相同的安全策略。UsernameTokenCallbackHandler 是与 ActAs 共享的实用程序。它为 OnBehalfOf 元素生成内容。最后，STS 中也添加了代码，OnBehalfOf 和 ActAs 共享了相同的代码。

A.8.3.1. Web 服务提供商

本节提供已更新的基本 WS-Trust 场景中的 Web 服务元素，以满足 OnBehalfOf 示例 的要求。这些组件包括：

- Web 服务提供商 WSDL
- Web 服务提供商接口
- Web 服务提供商实施
- OnBehalfOfCallbackHandler Class

A.8.3.1.1. Web 服务提供商 WSDL

OnBehalfOf Web 服务提供商的 WSDL 是 ws-provider 的 WSDL 的 克隆。wsp:Policy 部分是相同的。对服务端点、target Namespace、portType、绑定 名称和 服务进行了更新。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy" name="OnBehalfOfService"
          xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/onbehalfofwssecuritypolicy"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/"
          xmlns:wsp="http://www.w3.org/ns/ws-policy"
          xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
          xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
          xmlns:wsaws="http://www.w3.org/2005/08/addressing"
          xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
          xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy"
            schemaLocation="OnBehalfOfService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="OnBehalfOfServiceIface">
```

```
        <operation name="sayHello">
          <input message="tns:sayHello"/>
          <output message="tns:sayHelloResponse"/>
        </operation>
    </portType>
    <binding name="OnBehalfOfServicePortBinding" type="tns:OnBehalfOfServiceIface">
        <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
        <operation name="sayHello">
          <soap:operation soapAction=""/>
          <input>
             <soap:body use="literal"/>
             <wsp:PolicyReference URI="#Input_Policy" />
          </input>
          <output>
             <soap:body use="literal"/>
             <wsp:PolicyReference URI="#Output_Policy" />
          </output>
        </operation>
    </binding>
    <service name="OnBehalfOfService">
        <port name="OnBehalfOfServicePort" binding="tns:OnBehalfOfServicePortBinding">
          <soap:address location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
onbehalfof/OnBehalfOfService"/>
        </port>
    </service>
</definitions>
```

A.8.3.1.2. Web 服务提供商接口

OnBehalfOfServiceIface Web 服务提供商接口类是一个简单的 Web 服务定义。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
   targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy"
)
public interface OnBehalfOfServiceIface {
   @WebMethod
   String sayHello();
}
```

A.8.3.1.3. Web 服务提供商实施

OnBehalfOfServiceImpl Web 服务提供商实施类是一个简单的 POJO。它使用标准的 WebService 注释来定义服务端点和两个 Apache WSS4J 注释，即 EndpointProperties 和 EndpointProperty，用

于为 Apache CXF 运行时配置端点。提供的 WSS4J 配置信息用于 WSS4J 的 Crypto Merlin 实施。

OnBehalfOfService Impl 代表用户调用 ServiceImpl。setupService 方法执行所需的配置设置。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.*;
import java.util.Map;

@WebService
(
  portName = "OnBehalfOfServicePort",
  serviceName = "OnBehalfOfService",
  wsdlLocation = "WEB-INF/wsdl/OnBehalfOfService.wsdl",
  targetNamespace = "http://www.jboss.org/jbossws/ws-
extensions/onbehalfofwssecuritypolicy",
  endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfServiceIface"
)

@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof.OnBehalfOfCallbackHandler")
})

public class OnBehalfOfServiceImpl implements OnBehalfOfServiceIface {
  public String sayHello() {
    try {

      ServiceIface proxy = setupService();
      return "OnBehalfOf " + proxy.sayHello();

    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
```

```
      return null;
   }

   /**
    *
    * @return
    * @throws MalformedURLException
    */
   private  ServiceIface setupService()throws MalformedURLException {
      ServiceIface proxy = null;
      Bus bus = BusFactory.newInstance().createBus();

      try {
         BusFactory.setThreadDefaultBus(bus);

         final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() + ":8080/jaxws-
samples-wsse-policy-trust/SecurityService";
         final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wssecuritypolicy", "SecurityService");
         final URL wsdlURL = new URL(serviceURL + "?wsdl");
         Service service = Service.create(wsdlURL, serviceName);
         proxy = (ServiceIface) service.getPort(ServiceIface.class);

         Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
         ctx.put(SecurityConstants.CALLBACK_HANDLER, new OnBehalfOfCallbackHandler());

         ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "actasKeystore.properties" ));
         ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
         ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "../../META-INF/clientKeystore.properties" ));
         ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

         STSClient stsClient = new STSClient(bus);
         Map<String, Object> props = stsClient.getProperties();
         props.put(SecurityConstants.USERNAME, "bob");
         props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
         props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
         props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource(
            "actasKeystore.properties" ));
         props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

         ctx.put(SecurityConstants.STS_CLIENT, stsClient);

      } finally {
         bus.shutdown(true);
      }

      return proxy;
   }

}
```

A.8.3.1.4. OnBehalfOfCallbackHandler Class

OnBehalfOfCallbackHandler 是 WSS4J Crypto API 的回调处理程序。它用于在密钥存储中获取私钥的密码。此类使 Apache CXF 能够检索用于消息签名的用户名密码。此类已被更新，返回此服务的密码 myactaskey 和 OnBehalfOf 用户 alice。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.onbehalfof;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class OnBehalfOfCallbackHandler extends PasswordCallbackHandler {

  public OnBehalfOfCallbackHandler() {
    super(getInitMap());
  }

  private static Map<String, String> getInitMap() {
    Map<String, String> passwords = new HashMap<String, String>();
    passwords.put("myactaskey", "aspass");
    passwords.put("alice", "clarinet");
    passwords.put("bob", "trombone");
    return passwords;
  }

}
```

A.8.3.2. Web Service Requester

本节详细介绍了 WS -Trust 场景中的 ws-requester 元素，这些元素已更新，以满足 OnBehalfOf 示例 的要求。该组件为：

- OnBehalfOf Web 服务请求器实施类

A.8.3.2.1. OnBehalfOf Web 服务请求器实施类

客户端 OnBehalfOf ws-requester 使用标准步骤在前四行中创建对 Web 服务的引用。为满足端点安全要求，Web 服务的请求上下文使用 BindingProvider 配置。通过其提供消息生成所需的信息。此部分中声明了 OnBehalfOf 用户 alice，并向 STSClient 提供 callbackHandler、UsernameTokenCallbackHandler，用于生成 OnBehalfOf 消息元素的内容。在本例中，将创建一个 STSClient 对象，并提供给代理的请求上下文。另一种方法是提供带有 .it 后缀的键，如基本场景客户端中所执行的操作一样。OnBehalfOf 的使用通过 stsClient.setOnBehalfOf 调用方法进行配置。另一种方法是在 properties 映射中使用 key SecurityConstants.STS_TOKEN_ON_BEHALF_OF 和一个值。

```
final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
```

```
extensions/onbehalfofwssecuritypolicy", "OnBehalfOfService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
OnBehalfOfServiceIface proxy = (OnBehalfOfServiceIface)
service.getPort(OnBehalfOfServiceIface.class);


Bus bus = BusFactory.newInstance().createBus();
try {

   BusFactory.setThreadDefaultBus(bus);

   Map<String, Object> ctx = proxy.getRequestContext();

   ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
   ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

   // user and password OnBehalfOf user
   // UsernameTokenCallbackHandler will extract this information when called
   ctx.put(SecurityConstants.USERNAME,"alice");
   ctx.put(SecurityConstants.PASSWORD, "clarinet");

   STSClient stsClient = new STSClient(bus);

   // Providing the STSClient the mechanism to create the claims contents for OnBehalfOf
   stsClient.setOnBehalfOf(new UsernameTokenCallbackHandler());

   Map<String, Object> props = stsClient.getProperties();
   props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   props.put(SecurityConstants.ENCRYPT_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
   props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
   props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

   ctx.put(SecurityConstants.STS_CLIENT, stsClient);

} finally {
   bus.shutdown(true);
}
proxy.sayHello();
```

A.8.4. 场景：ActAs WS-Trust

ActA 功能用于需要复合委派的情况。它通常用于多层系统中，其中的应用代表登录用户调用服务，或者服务代表原始调用者调用其他服务。

actionas 不仅仅是 RequestSecurityToken (RST)中的新子元素。它在与 STS 协商令牌时提供有关原始调用者的额外信息。ActAs 元素通常采用令牌的形式，其身份声明（如名称、角色和授权代码）供客户端访问服务。

ActA 场景是基本 WS-Trust 场景的扩展。在本例中，ActAs 服务代表用户调用 ws-service。对基本场景的代码仅作几处补充。添加了 ActAs Web 服务提供商和回调处理程序。ActAs Web 服务的 WSDL 采用了与 ws-provider 相同的安全策略。UsernameTokenCallbackHandler 是一个新实用程序，可为 ActAs 元素生成内容。最后，STS 中添加了一些代码来支持 ActAs 请求。

A.8.4.1. Web 服务提供商

本节详细介绍了已更改的基本 WS-Trust 场景中的 Web 服务元素，以满足 ActAs 示例 的需求。这些组件包括：

- actas Web Service Provider WSDL
- actas Web 服务提供商接口
- actas Web 服务提供商实施
- ActAsCallbackHandler Class
- UsernameTokenCallbackHandler
- 加密属性和密钥存储文件
- 默认 MANIFEST.MF

A.8.4.1.1. Web 服务提供商 WSDL

ActAs Web 服务提供商的 WSDL 是 ws-provider 的 WSDL 的 克隆。wsp:Policy 部分是相同的。服务端点、target Namespace、port Type、绑定 名称和 服务 都有更改。

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
name="ActAsService"
             xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:wsp="http://www.w3.org/ns/ws-policy"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
             xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
             xmlns:wsaws="http://www.w3.org/2005/08/addressing"
             xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
             xmlns:t="http://docs.oasis-open.org/ws-sx/ws-trust/200512">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
          schemaLocation="ActAsService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="ActAsServiceIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="ActAsServicePortBinding" type="tns:ActAsServiceIface">
    <wsp:PolicyReference URI="#AsymmetricSAML2Policy" />
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
        <wsp:PolicyReference URI="#Input_Policy" />
      </input>
      <output>
        <soap:body use="literal"/>
        <wsp:PolicyReference URI="#Output_Policy" />
      </output>
    </operation>
  </binding>
  <service name="ActAsService">
    <port name="ActAsServicePort" binding="tns:ActAsServicePortBinding">
      <soap:address location="http://@jboss.bind.address@:8080/jaxws-samples-wsse-policy-trust-
actas/ActAsService"/>
    </port>
```

```
    </service>

</definitions>
```

A.8.4.1.2. Web 服务提供商接口

ActAsServiceIface Web 服务提供商界面类是一个简单的 Web 服务定义。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
(
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy"
)
public interface ActAsServiceIface {
  @WebMethod
  String sayHello();
}
```

A.8.4.1.3. Web 服务提供商实施

ActAsServiceImpl Web 服务提供商实施类是一个简单的 POJO。它使用标准的 WebService 注释来定义服务端点和两个 Apache WSS4J 注释（Ed pointProperties 和 EndpointProperty ），用于为 Apache CXF 运行时配置端点。提供的 WSS4J 配置信息用于 WSS4J 的 Crypto Merlin 实施。

ActAsServiceImpl 代表用户调用 ServiceImpl。setupService 方法执行所需的配置设置。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.STSClient;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.service.ServiceIface;
import org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared.WSTrustAppUtils;

import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Service;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
```

```
@WebService
(
   portName = "ActAsServicePort",
   serviceName = "ActAsService",
   wsdlLocation = "WEB-INF/wsdl/ActAsService.wsdl",
   targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/actaswssecuritypolicy",
   endpointInterface =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsServiceIface"
)

@EndpointProperties(value = {
      @EndpointProperty(key = "ws-security.signature.username", value = "myactaskey"),
      @EndpointProperty(key = "ws-security.signature.properties", value =
"actasKeystore.properties"),
      @EndpointProperty(key = "ws-security.encryption.properties", value =
"actasKeystore.properties"),
      @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas.ActAsCallbackHandler")
})

public class ActAsServiceImpl implements ActAsServiceIface {
   public String sayHello() {
      try {
         ServiceIface proxy = setupService();
         return "ActAs " + proxy.sayHello();
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
      return null;
   }

   private  ServiceIface setupService()throws MalformedURLException {
      ServiceIface proxy = null;
      Bus bus = BusFactory.newInstance().createBus();

      try {
         BusFactory.setThreadDefaultBus(bus);

         final String serviceURL = "http://" + WSTrustAppUtils.getServerHost() + ":8080/jaxws-
samples-wsse-policy-trust/SecurityService";
         final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/wssecuritypolicy", "SecurityService");
         final URL wsdlURL = new URL(serviceURL + "?wsdl");
         Service service = Service.create(wsdlURL, serviceName);
         proxy = (ServiceIface) service.getPort(ServiceIface.class);

         Map<String, Object> ctx = ((BindingProvider) proxy).getRequestContext();
         ctx.put(SecurityConstants.CALLBACK_HANDLER, new ActAsCallbackHandler());

         ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
         ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myactaskey" );
         ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
            Thread.currentThread().getContextClassLoader().getResource("../../META-
INF/clientKeystore.properties" ));
```

```
            ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myservicekey");

            STSClient stsClient = new STSClient(bus);
            Map<String, Object> props = stsClient.getProperties();
            props.put(SecurityConstants.USERNAME, "alice");
            props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
            props.put(SecurityConstants.STS_TOKEN_USERNAME, "myactaskey" );
            props.put(SecurityConstants.STS_TOKEN_PROPERTIES,

Thread.currentThread().getContextClassLoader().getResource("actasKeystore.properties" ));
            props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

            ctx.put(SecurityConstants.STS_CLIENT, stsClient);

        } finally {
            bus.shutdown(true);
        }

        return proxy;
    }

}
```

A.8.4.1.4. ActAsCallbackHandler Class

ActAsCallbackHandler 是 WSS4J Crypto API 的回调处理程序。它用于在密钥存储中获取私钥的密码。此类使 Apache CXF 能够检索用于消息签名的用户名密码。此类已更新，以返回此服务的密码，即 myactaskey 和 ActAs 用户 alice。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.actas;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;
import java.util.HashMap;
import java.util.Map;

public class ActAsCallbackHandler extends PasswordCallbackHandler {

    public ActAsCallbackHandler() {
        super(getInitMap());
    }

    private static Map<String, String> getInitMap() {
        Map<String, String> passwords = new HashMap<String, String>();
        passwords.put("myactaskey", "aspass");
        passwords.put("alice", "clarinet");
        return passwords;
    }
}
```

A.8.4.1.5. UsernameTokenCallbackHandler

RequestSecurityToken 的 ActAs 和 OnBeholdOf 子元素必须定义为 WSSE UsernameTokens。

此实用程序生成正确格式化的元素。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.shared;

import org.apache.cxf.helpers.DOMUtils;
import org.apache.cxf.message.Message;
import org.apache.cxf.ws.security.SecurityConstants;
import org.apache.cxf.ws.security.trust.delegation.DelegationCallback;
import org.apache.ws.security.WSConstants;
import org.apache.ws.security.message.token.UsernameToken;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import java.io.IOException;
import java.util.Map;

/**
* A utility to provide the 3 different input parameter types for jaxws property
* "ws-security.sts.token.act-as" and "ws-security.sts.token.on-behalf-of".
* This implementation obtains a username and password via the jaxws property
* "ws-security.username" and "ws-security.password" respectively, as defined
* in SecurityConstants.  It creates a wss UsernameToken to be used as the
* delegation token.
*/

public class UsernameTokenCallbackHandler implements CallbackHandler {

  public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
      if (callbacks[i] instanceof DelegationCallback) {
        DelegationCallback callback = (DelegationCallback) callbacks[i];
        Message message = callback.getCurrentMessage();

        String username =
          (String)message.getContextualProperty(SecurityConstants.USERNAME);
        String password =
          (String)message.getContextualProperty(SecurityConstants.PASSWORD);
        if (username != null) {
          Node contentNode = message.getContent(Node.class);
          Document doc = null;
          if (contentNode != null) {
            doc = contentNode.getOwnerDocument();
          } else {
            doc = DOMUtils.createDocument();
          }
          UsernameToken usernameToken = createWSSEUsernameToken(username,password,
doc);
          callback.setToken(usernameToken.getElement());
```

```
            }
          } else {
            throw new UnsupportedCallbackException(callbacks[i], "Unrecognized Callback");
          }
        }
      }

      /**
       * Provide UsernameToken as a string.
       * @param ctx
       * @return
       */
      public String getUsernameTokenString(Map<String, Object> ctx){
        Document doc = DOMUtils.createDocument();
        String result = null;
        String username = (String)ctx.get(SecurityConstants.USERNAME);
        String password = (String)ctx.get(SecurityConstants.PASSWORD);
        if (username != null) {
          UsernameToken usernameToken = createWSSEUsernameToken(username,password,
   doc);
          result = toString(usernameToken.getElement().getFirstChild().getParentNode());
        }
        return result;
      }

      /**
       *
       * @param username
       * @param password
       * @return
       */
      public String getUsernameTokenString(String username, String password){
        Document doc = DOMUtils.createDocument();
        String result = null;
        if (username != null) {
          UsernameToken usernameToken = createWSSEUsernameToken(username,password,
   doc);
          result = toString(usernameToken.getElement().getFirstChild().getParentNode());
        }
        return result;
      }

      /**
       * Provide UsernameToken as a DOM Element.
       * @param ctx
       * @return
       */
      public Element getUsernameTokenElement(Map<String, Object> ctx){
        Document doc = DOMUtils.createDocument();
        Element result = null;
        UsernameToken usernameToken = null;
          String username = (String)ctx.get(SecurityConstants.USERNAME);
        String password = (String)ctx.get(SecurityConstants.PASSWORD);
        if (username != null) {
          usernameToken = createWSSEUsernameToken(username,password, doc);
          result = usernameToken.getElement();
```

```
    }
    return result;
  }

  /**
   *
   * @param username
   * @param password
   * @return
   */
  public Element getUsernameTokenElement(String username, String password){
    Document doc = DOMUtils.createDocument();
    Element result = null;
    UsernameToken usernameToken = null;
    if (username != null) {
      usernameToken = createWSSEUsernameToken(username,password, doc);
      result = usernameToken.getElement();
    }
    return result;
  }

  private UsernameToken createWSSEUsernameToken(String username, String password,
Document doc) {

    UsernameToken usernameToken = new UsernameToken(true, doc,
      (password == null)? null: WSConstants.PASSWORD_TEXT);
    usernameToken.setName(username);
    usernameToken.addWSUNamespace();
    usernameToken.addWSSENamespace();
    usernameToken.setID("id-" + username);

    if (password != null){
      usernameToken.setPassword(password);
    }

    return usernameToken;
  }


  private String toString(Node node) {
    String str = null;

    if (node != null) {
      DOMImplementationLS lsImpl = (DOMImplementationLS)
        node.getOwnerDocument().getImplementation().getFeature("LS", "3.0");
      LSSerializer serializer = lsImpl.createLSSerializer();
      serializer.getDomConfig().setParameter("xml-declaration", false); //by default its true, so
set it to false to get String without xml-declaration
      str = serializer.writeToString(node);
    }
    return str;
  }

}
```

A.8.4.1.6. crypto 属性和密钥存储文件

ActA 服务必须提供自己的凭据。创建必要的 actasKeystore.properties 属性文件和 actasstore.jks 密钥存储。

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=aapass
org.apache.ws.security.crypto.merlin.keystore.alias=myactaskey
org.apache.ws.security.crypto.merlin.keystore.file=actasstore.jks
```

A.8.4.1.7. 默认 MANIFEST.MF

此应用程序需要访问 org.jboss.ws.cxf.jbossws-cxf 中提供的 JBossWS 和 Apache CXF API。在处理 ActAs 和 OnBehalfOf 扩展时，还需要 org.jboss.ws.cxf.sts 模块。dependency 语句指示服务器在部署时提供它们。

```
Manifest-Version: 1.0
Dependencies: org.jboss.ws.cxf.jbossws-cxf-client, org.jboss.ws.cxf.sts
```

A.8.4.2. 安全令牌服务

本节详细介绍了 WS-Trust 基本场景中的 STS 元素以满足 ActAs 示例的需求。这些组件包括：

- STS 实施类
- STSCallbackHandler Class

A.8.4.2.1. STS 实施类

通过地址扩展允许令牌收件人的声明接受 ActAs 地址和 OnBehalfOf 地址。地址指定为 reg-ex 模式。

TokenIsueOperation 要求提供 UsernameToken 验证器 类来验证 OnBehalfOf 的内容，并提供 UsernameTokenDelegationHandler 类，以处理 OnBehalfOf 用户的 ActAs 的令牌委派请求。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.Arrays;
import java.util.LinkedList;
```

```
import java.util.List;

import javax.xml.ws.WebServiceProvider;

import org.apache.cxf.annotations.EndpointProperties;
import org.apache.cxf.annotations.EndpointProperty;
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.sts.StaticSTSProperties;
import org.apache.cxf.sts.operation.TokenIssueOperation;
import org.apache.cxf.sts.operation.TokenValidateOperation;
import org.apache.cxf.sts.service.ServiceMBean;
import org.apache.cxf.sts.service.StaticService;
import org.apache.cxf.sts.token.delegation.UsernameTokenDelegationHandler;
import org.apache.cxf.sts.token.provider.SAMLTokenProvider;
import org.apache.cxf.sts.token.validator.SAMLTokenValidator;
import org.apache.cxf.sts.token.validator.UsernameTokenValidator;
import org.apache.cxf.ws.security.sts.provider.SecurityTokenServiceProvider;

@WebServiceProvider(serviceName = "SecurityTokenService",
    portName = "UT_Port",
    targetNamespace = "http://docs.oasis-open.org/ws-sx/ws-trust/200512/",
    wsdlLocation = "WEB-INF/wsdl/ws-trust-1.4-service.wsdl")
//dependency on org.apache.cxf module or on module that exports org.apache.cxf (e.g.
org.jboss.ws.cxf.jbossws-cxf-client) is needed, otherwise Apache CXF annotations are
ignored
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.username", value = "mystskey"),
    @EndpointProperty(key = "ws-security.signature.properties", value =
"stsKeystore.properties"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts.STSCallbackHandler"),
    @EndpointProperty(key = "ws-security.validate.token", value = "false") //to let the JAAS
integration deal with validation through the interceptor below
})
@InInterceptors(interceptors =
{"org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"})
public class SampleSTS extends SecurityTokenServiceProvider {
  public SampleSTS() throws Exception {
    super();

    StaticSTSProperties props = new StaticSTSProperties();
    props.setSignatureCryptoProperties("stsKeystore.properties");
    props.setSignatureUsername("mystskey");
    props.setCallbackHandlerClass(STSCallbackHandler.class.getName());
    props.setIssuer("DoubleItSTSIssuer");

    List<ServiceMBean> services = new LinkedList<ServiceMBean>();
    StaticService service = new StaticService();
    service.setEndpoints(Arrays.asList(
      "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
      "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",
      "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust/SecurityService",

      "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
      "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
      "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-actas/ActAsService",
```

```
         "http://localhost:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService",
         "http://\\[::1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-onbehalfof/OnBehalfOfService",
         "http://\\[0:0:0:0:0:0:0:1\\]:(\\d)*/jaxws-samples-wsse-policy-trust-
onbehalfof/OnBehalfOfService"
      ));
      services.add(service);

      TokenIssueOperation issueOperation = new TokenIssueOperation();
      issueOperation.setServices(services);
      issueOperation.getTokenProviders().add(new SAMLTokenProvider());
      // required for OnBehalfOf
      issueOperation.getTokenValidators().add(new UsernameTokenValidator());
      // added for OnBehalfOf and ActAs
      issueOperation.getDelegationHandlers().add(new UsernameTokenDelegationHandler());
      issueOperation.setStsProperties(props);

      TokenValidateOperation validateOperation = new TokenValidateOperation();
      validateOperation.getTokenValidators().add(new SAMLTokenValidator());
      validateOperation.setStsProperties(props);

      this.setIssueOperation(issueOperation);
      this.setValidateOperation(validateOperation);
   }
}
```

A.8.4.2.2. STSCallbackHandler 类

为 ActAs 示例添加用户 、 alice 和对应的密码。

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.trust.sts;

import java.util.HashMap;
import java.util.Map;

import org.jboss.wsf.stack.cxf.extensions.security.PasswordCallbackHandler;

public class STSCallbackHandler extends PasswordCallbackHandler {
   public STSCallbackHandler() {
      super(getInitMap());
   }

   private static Map<String, String> getInitMap() {
      Map<String, String> passwords = new HashMap<String, String>();
      passwords.put("mystskey", "stskpass");
      passwords.put("alice", "clarinet");
      return passwords;
   }
}
```

A.8.4.2.3. Web Service Requester

本节详细介绍了 WS-Trust 环境中的基本 WS -Trust 场景中的 ws-requester 元素，这些元素已更改为满足 ActAs 示例 的要求。该组件为：

- actas Web Service Requester 实施类

A.8.4.2.4. Web 服务请求者实施类

作为客户端 ActAs ws-requester 使用标准步骤在前四行中创建对 Web 服务的引用。为满足端点安全要求，Web 服务的请求上下文配置为使用 BindingProvider 来提供消息生成所需的信息。ActAs 用户 myactaskey 在此部分中声明，UsernameTokenCallbackHandler 用于向 STSClient 提供 ActAs 元素的内容。在本例中，将创建一个 STSClient 对象，并提供给代理的请求上下文。另一种方法是提供带有 .it 后缀的键，如基本场景客户端中所执行的操作一样。ActAs 的使用通过利用 SecurityConstants.STS_TOKEN_ACT_AS 键的属性映射进行配置。另一种方法是使用 STSClient.setActAs 方法。

```
final QName serviceName = new QName("http://www.jboss.org/jbossws/ws-
extensions/actaswssecuritypolicy", "ActAsService");
final URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ActAsServiceIface proxy = (ActAsServiceIface) service.getPort(ActAsServiceIface.class);

Bus bus = BusFactory.newInstance().createBus();
try {
   BusFactory.setThreadDefaultBus(bus);

   Map<String, Object> ctx = proxy.getRequestContext();

   ctx.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   ctx.put(SecurityConstants.ENCRYPT_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   ctx.put(SecurityConstants.ENCRYPT_USERNAME, "myactaskey");
   ctx.put(SecurityConstants.SIGNATURE_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   ctx.put(SecurityConstants.SIGNATURE_USERNAME, "myclientkey");

   // Generate the ActAs element contents and pass to the STSClient as a string
   UsernameTokenCallbackHandler ch = new UsernameTokenCallbackHandler();
   String str = ch.getUsernameTokenString("alice","clarinet");
   ctx.put(SecurityConstants.STS_TOKEN_ACT_AS, str);

   STSClient stsClient = new STSClient(bus);
   Map<String, Object> props = stsClient.getProperties();
   props.put(SecurityConstants.USERNAME, "bob");
   props.put(SecurityConstants.CALLBACK_HANDLER, new ClientCallbackHandler());
   props.put(SecurityConstants.ENCRYPT_PROPERTIES,
      Thread.currentThread().getContextClassLoader().getResource(
      "META-INF/clientKeystore.properties"));
   props.put(SecurityConstants.ENCRYPT_USERNAME, "mystskey");
   props.put(SecurityConstants.STS_TOKEN_USERNAME, "myclientkey");
```

```
    props.put(SecurityConstants.STS_TOKEN_PROPERTIES,
        Thread.currentThread().getContextClassLoader().getResource(
        "META-INF/clientKeystore.properties"));
    props.put(SecurityConstants.STS_TOKEN_USE_CERT_FOR_KEYINFO, "true");

    ctx.put(SecurityConstants.STS_CLIENT, stsClient);
} finally {
    bus.shutdown(true);
}
proxy.sayHello();
```

更新于 2024-02-08