JBoss Enterprise Application Platform 5

RESTEasy リファレンスガイド

JBoss Enterprise Application Platform 5 向け

エディッション 5.1.2

Last Updated: 2017-10-26

JBoss Enterprise Application Platform 5 RESTEasy リファレンスガイド

JBoss Enterprise Application Platform 5 向け
エディッション 5.1.2

Red Hat Documentation Group

法律上の通知

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

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, 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 Software Collections 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.

概要

本書は JBoss Enterprise Application Platform 5 およびそのパッチリリース向けの RESTEasy リファレンスガイドです。

目次

第1章 概要 **5**

第2章 インストール / 設定 **6**
2.1. ご使用のアプリケーションでの ESTEASY の使用 6
2.2. JAVAX.WS.RS.CORE.APPLICATION 8
2.3. RESTEASYLOGGING 9

第3章 @PATH、 @GET、 @POST などの使用 **11**
3.1. @PATH と正規表現のマッピング 11

第4章 @PATHPARAM **13**
4.1. 上級の @PATHPARAM および正規表現 13
4.2. @PATHPARAM と PATHSEGMENT 14

第5章 @QUERYPARAM **15**

第6章 @HEADERPARAM **16**

第7章 @MATRIXPARAM **17**

第8章 @COOKIEPARAM **18**

第9章 @FORMPARAM **19**

第10章 @FORM **20**

第11章 @DEFAULTVALUE **21**

第12章 @ENCODED とエンコーディング **22**

第13章 @CONTEXT **24**

第14章 JAX-RS リソースロケータとサブリソース **25**

第15章 JAX-RS コンテンツネゴシエーション **27**

第16章 コンテンツマーシャリング / プロバイダ **29**
16.1. デフォルトのプロバイダとデフォルトの JAX-RS コンテンツマーシャリング 29
16.2. @PROVIDER クラスでのコンテンツマーシャリング 29
16.3. プロバイダユーティリティクラス 29

第17章 JAXB プロバイダ **33**
17.1. JAXB デコレータ 33
17.2. プラグ可能な JAXBCONTEXT と CONTEXTRESOLVERS 35
17.3. JAXB および XML プロバイダ 35
17.3.1. @XmlHeader および @Stylesheet 35
17.4. JAXB および JSON プロバイダ 37
17.5. JAXB および FASTINFOSET プロバイダ 40
17.6. JAXB オブジェクトのアレイとコレクション 40
17.6.1. JSON および JAXB コレクション / アレイ 42
17.7. JAXB オブジェクトのマップ 43
17.7.1. JSON および JAXB マップ 45
17.7.2. Jettison プロバイダの考えられる問題 46
17.8. インターフェース、 抽象クラス、 JAXB 46

第18章 RESTEASY ATOM のサポート **47**
18.1. RESTEASY ATOM API とプロバイダ 47

18.2. ATOM プロバイダで JAXB を使用 48
18.3. APACHE ABDERA からの ATOM サポート 49
18.3.1. Abdera と Maven 49
18.3.2. Abdera プロバイダの使用 50

第19章 JACKSON による JSON サポート **54**
19.1. 考えられる JAXB プロバイダとの競合 55

第20章 マルチパートプロバイダ **56**
20.1. MULTIPART/MIXED による入力 56
20.2. JAVA.UTIL.LIST とマルチパートデータ 57
20.3. MULTIPART/FORM-DATA による入力 57
20.4. MULTIPART/FORM-DATA を使用した JAVA.UTIL.MAP 58
20.5. MULTIPART/RELATED による入力 58
20.6. マルチパートによる出力 59
20.7. マルチパート出力と JAVA.UTIL.LIST 60
20.8. MULTIPART/FORM-DATA による出力 60
20.9. JAVA.UTIL.MAP を使用したマルチパート FORMDATA 出力 61
20.10. MULTIPART/RELATED による出力 62
20.11. @MULTIPARTFORM と POJO 63
20.12. XOP (XML バイナリ最適パッケージ化) 64

第21章 YAML プロバイダ **66**

第22章 ストリングベース @*PARAM のストリングマーシャリング **67**

第23章 JAVAX.WS.RS.CORE.RESPONSE を使用した応答 **70**

第24章 例外処理 **71**
24.1. 例外マッパ 71
24.2. RESTEASY 内蔵の内部で送出された例外 72
24.3. RESTEASY 内蔵の例外をオーバーライドする 73

第25章 個別の JAX-RS リソース BEAN の設定 **74**

第26章 GZIP 圧縮 / 展開 **75**

第27章 RESTEASY キャッシング機能 **76**
27.1. @CACHE と @NOCACHE アノテーション 76
27.2. クライアント「ブラウザ」キャッシュ 76
27.3. ローカルのサーバー側応答キャッシュ 77

第28章 インターセプタ **80**
28.1. MESSAGEBODYREADER/WRITER インターセプタ 80
28.2. PREPROCESSINTERCEPTOR 82
28.3. POSTPROCESSINTERCEPTORS 83
28.4. CLIENTEXECUTIONINTERCEPTORS 83
28.5. バインディングインターセプタ 83
28.6. インターセプタの登録 84
28.7. インターセプタの順番と優先度 84
28.7.1. カスタム優先度 85

第29章 非同期 HTTP 要求処理 **88**
29.1. TOMCAT 6 と JBOSS 4.2.3 のサポート 89
29.2. SERVLET 3.0 のサポート 89
29.3. JBOSSWEB と JBOSS AS 5.0.X のサポート 90

第30章 非同期ジョブサービス 91
30.1. 非同期ジョブの使用 91
30.2. 一方的: ファイアアンドフォーゲット 92
30.3. 設定と構成 92

第31章 組み込みコンテナ 94

第32章 サーバー側疑似フレームワーク 95

第33章 JAX-RS と RESTEASY のセキュリティ 96

第34章 EJB 統合 98

第35章 SPRING の統合 100
35.1. 基本的な統合 100
35.2. SPRING MVC の統合 101

第36章 GUICE 1.0 の統合 103

第37章 クライアントフレームワーク 105
37.1. 抽象応答 106
37.2. クライアントとサーバー間でインターフェースを共有する 108
37.3. クライアントエラー処理 109
37.4. 手動の CLIENTREQUEST API 109

第38章 MAVEN と RESTEASY 110

第39章 JBOSS 5.X の統合 113

第40章 旧バージョンからの移行 114
40.1. 1.0.X および 1.1-RC1 からの移行 114

付録A 改訂履歴 115

第1章 概要

JSR-000311 JAX-RS は、 HTTP プロトコル上で RESTful Web サービスの Java API を提供する JCP (Java Community Process) 仕様です。 **RESTEasy** は、 すべての Servlet コンテナで実行できる JAX-RS 仕様の移植可能な実装で、 JBoss Application Server (JBoss AS) と緊密に統合し、JBoss AS 環境でユーザー体験を向上させます。 JAX-RS がサーバー側のみの仕様でも、 RESTEasy は RESTEasy JAX-RS クライアントフレームワークを利用して JAX-RS の機能をクライアント側へ提供できるため、 JAX-RS とインターフェースプロキシにて、送信する HTTP 要求をリモートサーバーへマップすることができます。

- RESTEasy には JAX-RS 実装が含まれています。
- RESTEasy は、 Java Development Kit 5.0 以上で実行される Tomcat やアプリケーションサーバーへ移植可能です。
- RESTEasy サーバー実装と Embedded JBoss を使用して junit のテストを行うことができます。
- RESTEasy は EJB (Enterprise JavaBeans) と Spring フレームワークを簡単に統合できます。
- RESTEasy には、 HTTP クライアントを書くプロセスを簡略化するクライアントフレームワークが含まれているため、 サーバーバインディング以外も定義することができます。

第2章 インストール / 設定

2.1. ご使用のアプリケーションでの ESTEASY の使用

RESTEasy は **WAR** アーカイブとしてデプロイされるため、 Servlet コンテナ内部にデプロイされなければなりません。

エンタープライズのコードソースから **resteasy-jaxrs-war** を取得することができます。ファイルはソースコードのアーカイブの **/resteasy** フォルダにあります。Red Hat Customer Support Portal からアーカイブをダウンロードします。

JAX-RS アノテーションが付けられたクラスのリソースやプロバイダを **/WEB-INF/lib** 内の 1 つまたは複数の **JAR** に置きます。別の方法として raw クラスファイルを **/WEB-INF/classes** 内に置きます。 デフォルトでは、 JAX-RS アノテーションが付けられたクラスに対するこれらのディレクトリ内で **JAR** やクラスをスキャンし、 システム内でデプロイ、登録するよう RESTEasy が設定されています。

RESTEasy は **ServletContextListener** と Servlet として実装され、 **WAR** ファイル内にデプロイされます。 **WEB-INF/web.xml** ファイルには以下が含まれています。

```
<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <!--Set this if you want Resteasy to scan for JAX-RS classes
    <context-param>
        <param-name>resteasy.scan</param-name>
        <param-value>true</param-value>
    </context-param>
    -->

    <!-- set this if you map the Resteasy servlet to something other than
/*
    <context-param>
       <param-name>resteasy.servlet.mapping.prefix</param-name>
       <param-value>/resteasy</param-value>
    </context-param>
    -->
    <!-- to turn on security
    <context-param>
        <param-name>resteasy.role.based.security</param-name>
        <param-value>true</param-value>
    </context-param>
    -->

    <listener>
        <listener-
class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listene
r-class>
    </listener>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</ser
vlet-class>
    </servlet>
```

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

ResteasyBootstrap リスナは RESTEasy の基本コンポーネントの一部を初期化し、**WAR** ファイルに存在するアノテーションクラスをスキャンします。 また、<context-param> エレメントより設定オプションも受け取ります。

RESTEasy Servlet の <servlet-mapping> に **/*** 以外の URL パターンがある場合、 これらの設定オプションを設定しなければなりません。 例えば、 以下が URL パターンであるとします。

```
<servlet-mapping>
  <servlet-name>Resteasy</servlet-name>
  <url-pattern>/restful-services/*</url-pattern>
</servlet-mapping>
```

この場合、***resteasy-servlet.mapping.prefix*** の値は以下でなければなりません。

```
<context-param>
  <param-name>resteasy.servlet.mapping.prefix</param-name>
  <param-value>/restful-services</param-value>
</context-param>
```

使用可能な <param-name> 値は以下の参照用に一覧表示された <param-value> デフォルト（または予想されるパターン）で概要が示されます。

使用可能な <context-param> パラメータ名

resteasy.servlet.mapping.prefix

/* でない場合、RESTEasy **servlet-mapping** の URL パターンを定義します。

resteasy.scan.providers

@Provider クラスをスキャンし、登録します。デフォルト値は **false** です。

resteasy.scan.resources

JAX-RS リソースクラスをスキャンします。デフォルト値は **false** です。

resteasy.scan

@Provider と JAX-RS リソースクラス (**@Path**、 **@GET**、 **@POST** など) の両方に対してスキャンを行い、登録します。

resteasy.providers

登録したい完全修飾 **@Provider** クラス名のコンマ区切りリストです。

resteasy.use.builtin.providers

デフォルトの内蔵 **@Provider** クラスが登録されているかを判断します。デフォルト値は **true** です。

resteasy.resources

登録したい完全修飾 JAX-RS リソースクラス名のコンマ区切りリストです。

resteasy.jndi.resources

JAX-RS リソースとして登録したいオブジェクトを参照する JNDI 名のコンマ区切りリストです。

javax.ws.rs.core.Application

特定のポータブルな方法でブートストラップを行うアプリケーションクラスの完全修飾名です。

ResteasyBootstrap リスナは **ResteasyProviderFactory** と **Registry** のインスタンスを設定します。 **ResteasyProviderFactory** インスタンスと **Registry** インスタンスは、**ServletContext** 属性である **org.jboss.resteasy.spi.ResteasyProviderFactory** と **org.jboss.resteasy.spi.Registry** より取得できます。

2.2. JAVAX.WS.RS.CORE.APPLICATION

javax.ws.rs.core.Application は、 デプロイメントの情報を提供するため実装できる標準的な JAX-RS クラスで、すべての JAX-RS ルートリソースとプロバイダをリストするクラスです。

```
/**
 * Defines the components of a JAX-RS application and supplies additional
 * metadata. A JAX-RS application or implementation supplies a concrete
 * subclass of this abstract class.
 */
public abstract class Application
{
   private static final Set<Object> emptySet = Collections.emptySet();

   /**
    * Get a set of root resource and provider classes. The default
lifecycle
    * for resource class instances is per-request. The default lifecycle
for
    * providers is singleton.
    * <p/>
    * <p>Implementations should warn about and ignore classes that do not
    * conform to the requirements of root resource or provider classes.
    * Implementations should warn about and ignore classes for which
    * {@link #getSingletons()} returns an instance. Implementations MUST
    * NOT modify the returned set.</p>
    *
    * @return a set of root resource and provider classes. Returning null
    *         is equivalent to returning an empty set.
    */
   public abstract Set<Class<?>> getClasses();

   /**
    * Get a set of root resource and provider instances. Fields and
properties
    * of returned instances are injected with their declared dependencies
    * (see {@link Context}) by the runtime prior to use.
    * <p/>
```

```
     * <p>Implementations should warn about and ignore classes that do not
     * conform to the requirements of root resource or provider classes.
     * Implementations should flag an error if the returned set includes
     * more than one instance of the same class. Implementations MUST
     * NOT modify the returned set.</p>
     * <p/>
     * <p>The default implementation returns an empty set.</p>
     *
     * @return a set of root resource and provider instances. Returning
null
     *         is equivalent to returning an empty set.
     */
    public Set<Object> getSingletons()
    {
       return emptySet;
    }

}
```

アプリケーションを使用するには、 アプリケーションを実装する完全修飾クラスを用いて Servlet **context-param** と **javax.ws.rs.core.Application** を設定する必要があります。 例は次の通りです。

```
    <context-param>
       <param-name>javax.ws.rs.core.Application</param-name>
       <param-value>com.mycom.MyApplicationConfig</param-value>
    </context-param>
```

この設定を行った場合、 クラスが重複登録されないようにするため自動スキャンを無効にしてください。

2.3. RESTEASYLOGGING

RESTEasy は **slf4j** を使用してさまざまなイベントをログに記録します。

slf4j API はロギング API の簡単なファサードとして機能することが目的で、 デプロイメント時に希望する実装をプラグインできるようにします。 デフォルトでは、 RESTEasy は **Apache log4j** を使用するよう設定されていますが、 slf4j によってサポートされるすべてのロギングプロバイダを使用可能です。

フレームワークに定義されている初期設定されたロギングカテゴリは次の通りです。 他のロギングカテゴリを追加することもできますが、 下記のロギングカテゴリを使用すると問題が発生した場合のトラブルシューティングが楽になります。

表2.1 ロギングカテゴリ

カテゴリ	機能
org.jboss.resteasy.core	コアの RESTEasy 実装によるすべてのアクティビティをログに記録します。

カテゴリ	機能
`org.jboss.resteasy.plugins.providers`	RESTEasy エントリプロバイダによるすべてのアクティビティをログに記録します。
`org.jboss.resteasy.plugins.server`	RESTEasy サーバー実装によるすべてのアクティビティをログに記録します。
`org.jboss.resteasy.specimpl`	JAX-RS 実装クラスによるすべてのアクティビティをログに記録します。
`org.jboss.resteasy.mock`	RESTEasy の模擬フレームワークによるすべてのアクティビティをログに記録します。

RESTEasy コードの開発を行っている場合、 **`LoggerCategories`** クラスはカテゴリ名や様々なロガーへのアクセスを容易にします。

第3章 @PATH、 @GET、 @POST などの使用

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

   @GET
   @Path("/books")
   public String getBooks() {...}

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

   @PUT
   @Path("/book/{isbn}")
   public void addBook(@PathParam("isbn") String id, @QueryParam("name")
String name) {...}

   @DELETE
   @Path("/book/{id}")
   public void removeBook(@PathParam("id") String id {...}


}
```

RESTEasy Servlet を設定し、 **`http://myhost.com/services`** のルートパスよりアクセスできるようにすると、 要求は **`Library`** クラスによって処理されます。

- GET http://myhost.com/services/library/books
- GET http://myhost.com/services/library/book/333
- PUT http://myhost.com/services/library/book/333
- DELETE http://myhost.com/services/library/book/333

`@javax.ws.rs.Path` アノテーションがクラスとリソースメソッドのいずれかか両方に存在しなければなりません。 クラスとメソッドの両方に存在する場合、 リソースメソッドへの相対パスはクラスとメソッドの連結になります。

`@javax.ws.rs package` には各 HTTP メソッドのアノテーションが含まれています (**`@GET`**、 **`@POST`**、 **`@PUT`**、 **`@DELETE`**、 **`@HEAD`**)。 アノテーションの HTTP メソッドへマップしたい公開メソッドにこれらのアノテーションを付けます。 クラス上に **`@Path`** アノテーションが存在する場合、**`@Path`** でマップしたいメソッドにアノテーションを付ける必要はありません。 他のメソッドを区別できれば複数の HTTP メソッドを使用することが可能です。

HTTP メソッドを適用せずにメソッドに **`@Path`** アノテーションを付けると、 アノテーションが付けられたメソッドは **`JAXRSResourceLocator`** と見なされます。

3.1. @PATH と正規表現のマッピング

@Path アノテーションは単純なパス表現のみに限定されません。 **@Path** の値に正規表現を挿入することもできます。 例は次の通りです。

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

   @GET
   @Path("{var:.*}/stuff")
   public String get() {...}
}
```

次の例では、 GET が **getResource()** メソッドへルーティングされます。

```
GET /resources/stuff
GET /resources/foo/stuff
GET /resources/on/and/on/stuff
```

表現のフォーマットは次の通りです。

```
"{" variable-name [ ":" regular-expression ] "}"
```

ここでは、 **regular-expression** は任意となります。 **regular-expression** の指定がないと、 下記のように表現は 1 つの特定セグメントのワイルドカードの一致がデフォルトとなります。

```
"([]*)"
```

例は次の通りです。

```

@Path("/resources/{var}/stuff")
```

上記の例は以下に一致します。

```
GET /resources/foo/stuff
GET /resources/bar/stuff
```

しかし、 下記には一致しません。

```
GET /resources/a/bunch/of/stuff
```

第4章 @PATHPARAM

@PathParam は変数 URI パスフラグメントをメソッド呼び出しへマップできるようにします。

```
@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
   }
}
```

これにより、 リソースの URI に変数 ID を組み込むことができます。 前述の例を見ると、 アクセスしたい book の情報を **isbn** URI パラメータが渡していることが分かります。 プリミティブパラメータタイプ、 ストリング、 ストリングパラメータを取る Java オブジェクト、またはストリングパラメータを取る静的 **valueOf** メソッドを挿入することができます。 例えば、 **isbn** を真のオブジェクトにしたい場合、 次を書き込みます。

```


   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") ISBN id) {...}


   public class ISBN {
      public ISBN(String str) {...}
   }
```

また、 公開ストリングコンストラクタの代わりに **valueOf** メソッドを使用することもできます。

```
  public class ISBN {

     public static ISBN valueOf(String isbn) {...}
  }
```

4.1. 上級の @PATHPARAM および正規表現

さらに複雑な **@PathParam** の使用方法もあります。

1 つの URI セグメントに組み込まれた 1 つまたは複数の **@PathParam** を指定することができます。

```
1. @Path("/aaa{param}bbb")
2. @Path("/{name}-{zip}")
3. @Path("/foo{name}-{zip}bar")
```

フォーム "/aaa111bbb" の URI は指定された最初のパラメータと一致します。 "/bill-02115" は 2 つ目のパラメータ、 "foobill-02115bar" は 3 つ目のパラメータと一致します。

「@Path と正規表現のマッピング」 では、 次のように **@Path** 値内で正規表現が使用できることを説明しました。

```
@GET
@Path("/aaa{param:b+}/{many:.*}/stuff")
public String getIt(@PathParam("param") String bs, @PathParam("many")
String many) {...}
```

このように **@Path** が定義されると、 要求 **GET /aaabb/some/stuff** は **bb** の **"param"** 値と、**some** の **"many"** 値を持つようになります。 要求 **GET /aaab/a/lot/of/stuff** は **"param"** が値の **b** と **"many"** が値の **a/lot/of** を持つようになります。

4.2. @PATHPARAM と PATHSEGMENT

仕様には、 **javax.ws.rs.core.PathSegment** で呼び出された URI パスのフラグメントを検証するための大変単純な抽象化があります。

```
public interface PathSegment {

    /**
     * Get the path segment.
     * <p>
     * @return the path segment
     */
    String getPath();
    /**
     * Get a map of the matrix parameters associated with the path segment
     * @return the map of matrix parameters
     */
    MultivaluedMap<String, String> getMatrixParameters();

}
```

RESTEasy は、 **@PathParam** の値の代わりに **PathSegment** を挿入することができます。

```
   @GET
   @Path("/book/{id}")
   public String getBook(@PathParam("id") PathSegment id) {...}
```

これは、**マトリックスパラメータ** を使用する複数の **@PathParam** がある場合に便利です。 マトリックスパラメータは、 URI パスセグメントに組み込まれた名前と値のペアの任意セットです。 **PathSegment** オブジェクトによって、 これらのパラメータへアクセスすることができます。 詳細は 7 章*@MatrixParam* を参照してください。

マトリックスパラメータの例は次の通りです。

```
GET http://host.com/library/book;name=EJB 3.0;author=Bill Burke
```

マトリックスパラメータは、属性や raw ID が対応できるリソースを表します。

第5章 @QUERYPARAM

@QueryParam は、 URI クエリストリングパラメータや URL フォームのエンコードされたパラメータをメソッド呼び出しへマップできるようにします。

```
GET /books?num=5
```

```
@GET
public String getBooks(@QueryParam("num") int num) {
...
}
```

RESTEasy はサーブレット上で構築されるため、 URL クエリストリングと URL フォームのエンコードされたパラメータを区別することができません。 **`@PathParam`** と同様、パラメータのタイプはプリマティブ、ストリング、またはストリングコンストラクタあるいは静的 **`valueOf()`** メソッドを持つクラスのいずれかになります。

第6章 @HEADERPARAM

@HeaderParam アノテーションは要求 HTTP ヘッダをメソッド呼び出しへマップできるようにします。

```
GET /books?num=5
```

```
@GET
public String getBooks(@HeaderParam("From") String from) {
...
}
```

@PathParam と同様に、 パラメータタイプはプリミティブ、 ストリング、またはストリングコンストラクタあるいは静的 **valueOf()** メソッドを持つクラスのいずれかになります。 例えば、 **MediaType** には **valueOf()** メソッドがあるため、 以下を実行することができます。

```
@PUT
public void put(@HeaderParam("Content-Type") MediaType contentType,
...)
```

第7章 @MATRIXPARAM

マトリックスパラメータは URI パスセグメントに組み込まれた名前と値のペアの任意セットになります。 次はマトリックスパラメータの例です。

```
GET http://host.com/library/book;name=EJB 3.0;author=Bill Burke
```

マトリックスパラメータは、 属性や raw ID が対応できるリソースを表します。 **@MatrixParam** アノテーションは URI マトリックスパラメータをメソッド呼び出しへ挿入できるようにします。

```
   @GET
   public String getBook(@MatrixParam("name") String name,
@MatrixParam("author") String author) {...}
```

現バージョンの仕様が解決しないことは **@MatrixParam** の制限となります。 例えば、 異なるパスセグメントに同じ **MatrixParam** が同時に存在する場合、 現時点では **PathParam** と **PathSegment** を組み合わせて使用することが推奨されます。

第8章 @COOKIEPARAM

@CookieParam アノテーションはクッキーの値や HTTP 要求クッキーのオブジェクト表現をメソッド呼び出しへ挿入できるようにします。

```
GET /books?num=5
```

```
@GET
public String getBooks(@CookieParam("sessionid") int id) {
...
}

@GET
public String getBooks(@CookieParam("sessionid")
javax.ws.rs.core.Cookie id) {...}
```

@PathParam と同様に、 パラメータタイプはプリミティブ、 ストリング、またはストリングコンストラクタあるいは静的 **valueOf()** メソッドを持つクラスのいずれかになります。 **javax.ws.rs.core.Cookie** クラスよりクッキーのオブジェクト表現を取得することもできます。

第9章 @FORMPARAM

入力要求ボディのタイプが **application/x-www-form-urlencoded** (HTML フォーム) の場合、 個別のフォームパラメータを要求ボディよりメソッドパラメータ値へ挿入することができます。

```
<form method="POST" action="/resources/service">
First name:
<input type="text" name="firstname">
<br>
Last name:
<input type="text" name="lastname">
</form>
```

このフォームを使用して投稿する場合、 サービスは次のようになります。

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

   @Path("/resources/service")
   @POST
   public void addName(@FormParam("firstname") String first,
@FormParam("lastname") String last) {...}
```

MultivaluedMap<String, String> へアンマーシャルするデフォルトの **application/x-www-form-urlencoded** と **@FormParam** を組み合わせることはできません。 以下は違法となります。

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

   @Path("/resources/service")
   @POST
   @Consumes("application/x-www-form-urlencoded")
   public void addName(@FormParam("firstname") String first,
MultivaluedMap<String, String> form) {...}
```

第10章 @FORM

これは挿入されたクラス内で **@*Param** アノテーションの再使用を許可する RESTEasy 固有のアノテーションです。 RESTEasy はクラスをインスタンス化し、 アノテーションが付けられた **@*Param** や **@Context** プロパティへ値を挿入します。 メソッドにある多くのパラメータを 1 つの値オブジェクトへ圧縮したい場合に便利です。

```
public class MyForm {

   @FormParam("stuff")
   private int stuff;


   @HeaderParam("myHeader")
   private String header;


   @PathParam("foo")
   public void setFoo(String foo) {...}
}



@POST
@Path("/myservice")
public void post(@Form MyForm form) {...}
```

/myservice へ投稿があると、 RESTEasy は **MyForm** のインスタンスをインスタンス化し、 フォームパラメータ **stuff** を **stuff** フィールド、 ヘッダ **myheader** を **header** フィールドに挿入します。 そして、 **foo** の **@PathParam** 変数で **setFoo** メソッドを呼び出します。

第11章 @DEFAULTVALUE

@DefaultValue は、 他の **@*Param** アノテーションと組み合わせ、 HTTP 要求項目が存在しない場合にデフォルト値を定義できるパラメータアノテーションです。

```
    @GET
    public String getBooks(@QueryParam("num") @DefaultValue("10") int num)
{...}
```

第12章 @ENCODED とエンコーディング

JAX-RS は、 エンコードまたはデコードされた **@*Param** を取得し、 エンコードまたはデコードされたストリングを使用してパス定義やパラメータ名を指定します。

@javax.ws.rs.Encoded アノテーションは、 クラスやメソッド、 パラメータに使用することができます。 デフォルトでは、 挿入された **@PathParam** と **@QueryParam** がデコードされます。 **@Encoded** アノテーションを追加すると、 パラメータの値はエンコードされた形式で提供されます。

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

  @Path("/{param}")
  @GET
  public String get(@PathParam("param") @Encoded String param) {...}
```

前述の例では、 **get()** メソッドの **param** に挿入された **@PathParam** の値は URL エンコードされます。 **@Encoded** アノテーションをパラメータアノテーションとして追加すると、この動作を引き起こします。

メソッド全体で **@Encoded** アノテーションを使用して **@QueryParam** や **@PathParam** の値の組み合わせをエンコードすることもできます。

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

   @Path("/{param}")
   @GET
   @Encoded
   public String get(@QueryParam("foo") String foo, @PathParam("param")
String param) {}
}
```

この例では、 **foo** クエリパラメータと **param** パスパラメータの値がエンコードされた値として挿入されます。

クラス全体がエンコードされるようデフォルトを設定することもできます。

```
@Path("/")
@Encoded
public class ClassEncoded {

   @GET
   public String get(@QueryParam("foo") String foo) {}
}
```

@Path アノテーションは **encode** と呼ばれる属性を持っています。 この属性は、 提供された値のリテラル部分 (テンプレート変数の一部でない文字) が URL エンコードされるか制御します。 **true** の場合、 URL テンプレートにある無効な文字は自動的にエンコードされます。 **false** の場合、 全ての文字が URI で有効な文字でなければなりません。 デフォルトでは、 **encode** 属性は **true** に設定されています (手作業で文字をエンコードすることもできます)。

```
@Path(value="hello%20world", encode=false)
```

`@Path.encode()` は、 要求のクエリパラメータを検索する前に、 指定されたクエリパラメータ名をコンテナがエンコードするべきであるかを制御します。

```
@QueryParam(value="hello%20world", encode=false)
```

第13章 @CONTEXT

`@Context` アノテーションは、 **`javax.ws.rs.core.HttpHeaders`**、 **`javax.ws.rs.core.UriInfo`**、 **`javax.ws.rs.core.Request`**、 **`javax.servlet.HttpServletRequest`**, **`javax.servlet.HttpServletResponse`**、 **`javax.servlet.ServletConfig`**、 **`javax.servlet.ServletContext`**、 **`javax.ws.rs.core.SecurityContext`** オブジェクトのインスタンスを挿入できるようにします。

第14章 JAX-RS リソースロケータとサブリソース

リソースクラスは要求を部分的に処理し、 別の **サブリソース** オブジェクトを提供して要求の未処理部分を処理します。 例は次の通りです。

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

   @Path("/customers/{id}")
   public Customer getCustomer(@PathParam("id") int id) {
      Customer cust = ...; // Find a customer object
      return cust;
   }
}


public class Customer {

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

    @Path("/address")
    public String getAddress() {...}

}
```

@Path アノテーションを持ち、 HTTP メソッドがないリソースメソッドは **サブリソースロケータ** と考えられます。 サブリソースロケータは、 要求を処理できるオブジェクトを提供します。 前述のコード例では、**ShoppingStore** がルートリソースとなります。そのクラスには **@Path** アノテーションが付けられているためです。**getCustomer()** はサブリソースロケータメソッドとなります。

クライアントが以下を呼び出したとします。

```
GET /customer/123
```

この場合、 **ShoppingStore.getCustomer()** メソッドが最初に呼び出されます。 このメソッドは、要求に対応する **Customer** オブジェクトを提供します。 HTTP 要求は **Customer.get()** メソッドへ送信されます。 別の例を見てみましょう。

```
GET /customer/123/address
```

この要求では、同じく最初に **ShoppingStore.getCustomer()** メソッドが呼び出されます。**Customer** オブジェクトが返され、 残りの要求は **Customer.getAddress()** メソッドへ送信されます。

サブリソースロケータには、 要求の送信方法を決定するため、 ロケータメソッドの結果がランタイム時に動的に処理される興味深い機能もあります。 このため、 **ShoppingStore.getCustomer()** メソッドは特定のタイプを宣言する必要がありません。

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

   @Path("/customers/{id}")
```

```
   public java.lang.Object getCustomer(@PathParam("id") int id) {
      Customer cust = ...; // Find a customer object
      return cust;
   }
}


public class Customer {

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

    @Path("/address")
    public String getAddress() {...}

}
```

前述の例では、 **getCustomer()** は **java.lang.Object** を返します。 ランタイム時、 JAX-RS サーバーは要求ごとに **getCustomer()** が返すオブジェクトを基に要求の送信方法を決定します。

例えば、 顧客に対するクラス階層があるとします。 **Customer** は抽象ベースで、 **CorporateCustomer** と **IndividualCustomer** はサブクラスです。 この場合、 **getCustomer()** メソッドはクエリするクラスや返されるコンテンツを理解する必要がなく Hibernate 多形クエリを実行することがあります。

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

   @Path("/customers/{id}")
   public java.lang.Object getCustomer(@PathParam("id") int id) {
      Customer cust = entityManager.find(Customer.class, id);
      return cust;
   }
}


public class Customer {

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

    @Path("/address")
    public String getAddress() {...}

}

public class CorporateCustomer extendsCustomer {

    @Path("/businessAddress")
    public String getAddress() {...}

}
```

第15章 JAX-RS コンテンツネゴシエーション

HTTP プロトコルには、 クライアントとサーバーが転送するコンテンツのタイプや受信を希望するコンテンツタイプを指定できるようにする内蔵のコンテンツネゴシエーションヘッダがあります。 サーバーは **@Produces** および **@Consumes** ヘッダより希望するコンテンツを宣言します。

@Consumes は特定のリソースやリソースメソッドが消費するメディアタイプのアレイです。 例は次の通りです。

```
@Consumes("text/*")
@Path("/library")
public class Library {

@POST
public String stringBook(String book) {...}


@Consumes("text/xml")
@POST
public String jaxbBook(Book book) {...}
```

クライアントが要求を行うと、 JAX-RS は最初にパスと一致するすべてのメソッドを検索します。 その後、 クライアントが送信したコンテンツタイプヘッダを基にオブジェクトを仕分けします。 クライアントが以下を送信するとしましょう。

```
POST /library
content-type: text/plain

this is a nice book
```

この場合、 デフォルトの **text/*** メディアタイプと一致する **stringBook()** メソッドが呼び出されます。 クライアントが次のような XML を送信するとしましょう。

```
POST /library
content-type: text/xml

<book name="EJB 3.0" author="Bill Burke"/>
```

この場合、 **jaxbBook()** が呼び出されます。

@Produces は、 クライアント要求をマップし、 その要求をクライアントの **Accept** ヘッダと一致するために使用されます。 **Accept HTTP** ヘッダはクライアントによって送信され、 クライアントがサーバーから受信を希望するメディアタイプを定義します。

```
@Produces("text/*")
@Path("/library")
public class Library {

@GET
@Produces("application/json")
public String getJSON() {...}
```

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

クライアントが以下を送信するとしましょう。

```
        GET /library
        Accept: application/json
```

この場合、 **`getJSON()`** メソッドが呼び出されます。

`@Consumes` と **`@Produces`** が複数のメディアタイプをサポートするにはそれらをリストで表します。 クライアントの **`Accept`** ヘッダも受信する複数のメディアタイプをリストできます。 特定されたメディアタイプが最初に選択されます。**`Accept`** ヘッダ (または **`@Produces`** や **`@Consumes`**) は、要求をリソースメソッドと一致させる加重選好 (weighted preference) を指定することもできます (RFC 2616 の14.1 項に説明されています)。 RESTEasy は、コンテンツネゴシエーションのこのような複雑なメソッドのサポートを提供します。

JAX-RS によって使用される代替のメソッドは、 etag や最終変更ヘッダ、 その他の事前状態に加え、メディアタイプやコンテンツ言語、 コンテンツエンコーディングの組み合わせとなります。 これはさらに複雑な形式のコンテンツネゴシエーションで、 **`javax.ws.rs.Variant`** や **`VarianListBuilder`**、 **`Request`** オブジェクトを介しアプリケーション開発者がプログラムを用いて実行します。 **`Request`** は **`@Context`** アノテーションを使用して挿入されます (詳細は JavaDoc を参照してください)。

第16章 コンテンツマーシャリング / プロバイダ

16.1. デフォルトのプロバイダとデフォルトの JAX-RS コンテンツマーシャリング

RESTEasy は複数のメッセージボディタイプを自動的にマーシャリングおよびアンマーシャリングすることができます。

表16.1 メッセージボディタイプ

メディアタイプ	Java タイプ
application/*+xml、 text/*+xml、 application/*+json、 application/*+fastinfoset、 application/atom+*	JaxB アノテーションが付けられたクラス
/	java.lang.String
/	java.io.InputStream
text/plain	primitives、 java.lang.String、または String コンストラクタをあるいは入力の静的 valueOf(String) メソッド、出力の toString() を持つタイプすべて
/	javax.activation.DataSource
/	java.io.File
/	byte[]
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

16.2. @PROVIDER クラスでのコンテンツマーシャリング

JAX-RS 仕様は、 要求ボディまたは応答ボディのリーダーとライターのプラグインを可能にします。 実行するには、 クラスに **`@Provider`** アノテーションを付け、 リーダーの **`@Produces`** タイプを指定します。 **`MessageBodyReader`** と **`MessageBodyWriter`** インターフェースも実装する必要があります。

16.3. プロバイダユーティリティクラス

`javax.ws.rs.ext.Providers` は、 **`MessageBodyReader`**、 **`MessageBodyWriter`**、 **`ContextResolver`**、 **`ExceptionMapper`** の場所を特定できるようにする簡単な挿入可能インターフェースです。 これにより、 マルチパートプロバイダ (他のコンテンツタイプを組み込むコンテンツタイプ) の実装も可能になります。

```
public interface Providers
{

   /**
```

```
     * Get a message body reader that matches a set of criteria. The set of
     * readers is first filtered by comparing the supplied value of
     * {@code mediaType} with the value of each reader's
     * {@link javax.ws.rs.Consumes}, ensuring the supplied value of
     * {@code type} is assignable to the generic type of the reader, and
     * eliminating those that do not match.
     * The list of matching readers is then ordered with those with the
best
     * matching values of {@link javax.ws.rs.Consumes} (x/y > x/* > */*)
     * sorted first. Finally, the
     * {@link MessageBodyReader#isReadable}
     * method is called on each reader in order using the supplied criteria
and
     * the first reader that returns {@code true} is selected and returned.
     *
     * @param type        the class of object that is to be written.
     * @param mediaType   the media type of the data that will be read.
     * @param genericType the type of object to be produced. E.g. if the
     *                    message body is to be converted into a method
parameter, this will be
     *                    the formal type of the method parameter as
returned by
     *                    <code>Class.getGenericParameterTypes</code>.
     * @param annotations an array of the annotations on the declaration of
the
     *                    artifact that will be initialized with the
produced instance. E.g. if the
     *                    message body is to be converted into a method
parameter, this will be
     *                    the annotations on that parameter returned by
     *                    <code>Class.getParameterAnnotations</code>.
     * @return a MessageBodyReader that matches the supplied criteria or
null
     *         if none is found.
     */
    <T> MessageBodyReader<T> getMessageBodyReader(Class<T> type,
                                                  Type genericType,
Annotation annotations[], MediaType mediaType);

    /**
     * Get a message body writer that matches a set of criteria. The set of
     * writers is first filtered by comparing the supplied value of
     * {@code mediaType} with the value of each writer's
     * {@link javax.ws.rs.Produces}, ensuring the supplied value of
     * {@code type} is assignable to the generic type of the reader, and
     * eliminating those that do not match.
     * The list of matching writers is then ordered with those with the
best
     * matching values of {@link javax.ws.rs.Produces} (x/y > x/* > */*)
     * sorted first. Finally, the
     * {@link MessageBodyWriter#isWriteable}
     * method is called on each writer in order using the supplied criteria
and
     * the first writer that returns {@code true} is selected and returned.
     *
     * @param mediaType   the media type of the data that will be written.
```

```
     * @param type        the class of object that is to be written.
     * @param genericType the type of object to be written. E.g. if the
     *                    message body is to be produced from a field,
this will be
     *                    the declared type of the field as returned by
     *                    <code>Field.getGenericType</code>.
     * @param annotations an array of the annotations on the declaration of
the
     *                    artifact that will be written. E.g. if the
     *                    message body is to be produced from a field,
this will be
     *                    the annotations on that field returned by
     *                    <code>Field.getDeclaredAnnotations</code>.
     * @return a MessageBodyReader that matches the supplied criteria or
null
     *         if none is found.
     */
    <T> MessageBodyWriter<T> getMessageBodyWriter(Class<T> type,
                                                  Type genericType,
Annotation annotations[], MediaType mediaType);

    /**
     * Get an exception mapping provider for a particular class of
exception.
     * Returns the provider whose generic type is the nearest superclass of
     * {@code type}.
     *
     * @param type the class of exception
     * @return an {@link ExceptionMapper} for the supplied type or null if
none
     *         is found.
     */
    <T extends Throwable> ExceptionMapper<T> getExceptionMapper(Class<T>
type);

    /**
     * Get a context resolver for a particular type of context and media
type.
     * The set of resolvers is first filtered by comparing the supplied
value of
     * {@code mediaType} with the value of each resolver's
     * {@link javax.ws.rs.Produces}, ensuring the generic type of the
context
     * resolver is assignable to the supplied value of {@code contextType},
and
     * eliminating those that do not match. If only one resolver matches
the
     * criteria then it is returned. If more than one resolver matches then
the
     * list of matching resolvers is ordered with those with the best
     * matching values of {@link javax.ws.rs.Produces} (x/y > x/* > */*)
     * sorted first. A proxy is returned that delegates calls to
     * {@link ContextResolver#getContext(java.lang.Class)} to each matching
context
     * resolver in order and returns the first non-null value it obtains or
null
```

```
     * if all matching context resolvers return null.
     *
     * @param contextType the class of context desired
     * @param mediaType   the media type of data for which a context is
required.
     * @return a matching context resolver instance or null if no matching
     *         context providers are found.
     */
   <T> ContextResolver<T> getContextResolver(Class<T> contextType,
                                             MediaType mediaType);
}
```

次のように、 **MessageBodyReader** や **MessageBodyWriter** に **Providers** のインスタンスを挿入できます。

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

            private @Context Providers providers;

            ...

            }
```

第17章 JAXB プロバイダ

RESTEasy には JAXB アノテーションが付けられたクラスのマーシャリングやアンマーシャリングのサポートが含まれています。 RESTEasy には複数の JAXB プロバイダが含まれており、 XJC によって生成されたクラスと **@XmlRootElement** アノテーションが付けられたクラスとの若干の相違に対応するか、 直接 **JAXBElement** クラスに対応します。

開発で JAX-RS API を使用すると、 呼び出されるプロバイダは透過的に選択されます。 本章では、プロバイダに **直接** アクセスしたい場合に最適な設定の数々について説明します。

パラメータタイプ (戻りタイプ) が JAXB アノテーションが付いているオブジェクト (**@XmlRootEntity** や **@XmlType** など) か **JAXBElement** であると、 RESTEasy は JAXB プロバイダを選択します。 リソースクラス (リソースメソッド) には **@Consumes** または **@Produces** アノテーションが付けられ、 次の値の 1 つ以上が含まれます。

- **text/*+xml**
- **application/*+xml**
- **application/*+fastinfoset**
- **application/*+json**

RESTEasy はリソースで使用される戻りタイプを基に異なるプロバイダを選択します。 本項では選択処理の仕組みについて説明します。

@XmlRootElement アノテーションが付けられたクラスは **JAXBXmlRootElementProvider** によって処理されます。 このプロバイダは、 カスタム JAXB エンティティの基本的なマーシャリングとアンマーシャリングを処理します。

通常、XJC によって生成されるクラスには **@XmlRootElement** アノテーションが含まれていません。マーシャリングされるようにするには、 **JAXBElement** のインスタンスでラッピングされなければなりません。 通常、 **XmlRegistry** として動作する **ObjectFactory** というメソッドをクラス上で呼び出すことになります。

クラスに **XmlType** アノテーションが付けられ、 **XmlRootElement** アノテーションが付けられていない場合に **JAXBXmlTypeProvider** プロバイダが選択されます。 このプロバイダはターゲットクラスの **XmlRegistry** を探します。 デフォルトでは、 JAXB 実装はターゲットクラスと同じパッケージにある **ObjectFactory** というクラスを作成します。 **ObjectFactory** には、 オブジェクトインスタンスをパラメータとする **create** メソッドが含まれています。 例えば、 ターゲットタイプが **Contact** の場合、 **ObjectFactory** には 1 つのメソッドがあります。

```
public JAXBElement createContact(Contact value) {..
```

リソースが **JAXBElement** クラスと直接動作する場合、 RESTEasy ランタイムは **JAXBElementProvider** を選択します。 このプロバイダは、 適切な **JAXBContext** を選択するため、 **JAXBElement** の **ParameterizedType** 値を検証します。

17.1. JAXB デコレータ

RESTEasy の JAXB プロバイダは **Marshaller** インスタンスと **Unmarshaller** インスタンスをデコレーションすることができます。 **Marshaller** や **Unmarshaller** のデコレーションを引き起こすアノテーションを追加します。 デコレータは、 **Marshaller** プロパティや **Unmarshaller** プロパティの設定、 検証設定などのタスクを実行することができます。

例えば、 XML 文書の **pretty-printing** をトリガするアノテーションを作成したい場合、 raw JAXB で **`Marshaller.JAXB_FORMATTED_OUTPUT`** の **`Marshaller`** にプロパティを設定しますが、 この代わりに **マーシャラデコレータ** を書くことにします。

最初にアノテーションを定義します。

```
 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 {}
```

動作するようにするには、 **`@Pretty`** アノテーションに **`@Decorator`** と呼ばれるメタアノテーションを付ける必要があります。 **`target()`** 属性は JAXB **`Marshaller`** クラスでなければなりません。 次に、 **`processor()`** 属性クラスを書きます。

```
 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;

 /**
  * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
  * @version $Revision: 1 $
  */
 @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);
    }
 }
```

`processor` 実装は **`DecoratorProcessor`** インターフェースを実装しなければなりません。 また、 **`@DecorateTypes`** アノテーションが付けられている必要があります。 このアノテーションはプロセッサと動作するメディアタイプを指定します。

これでアノテーションと **`Processor`** が定義されたため 、JAX-RS リソースメソッドや JAXB タイプで以下のように使用できるようになりました。

```
   @GET
   @Pretty
   @Produces("application/xml")
   public SomeJAXBObject get() {...}
```

分かりにくい場合は、 RESTEasy ソースコードをチェックして **@XmlHeader** の実装について確認してください。

17.2. プラグ可能な JAXBCONTEXT と CONTEXTRESOLVERS

本機能は関係する原理を理解している方のみが使用するようにしてください。

デフォルトでは、 RESTEasy はマーシャリングまたはアンマーシャリングするクラスに応じて、 クラスタイプごとに **JAXBContext** インスタンスを作成しキャッシュします。 RESTEasy が**JAXBContext** を作成しないようにするには、 **javax.ws.rs.ext.ContextResolver** のインスタンスを実装して独自にプラグインできるようにします。

```
 public interface ContextResolver<T>
 {
T getContext(Class<?> type);
 }

 @Provider
 @Produces("application/xml")
 public class MyJAXBContextResolver implements
ContextResolver<JAXBContext>
 {
JAXBContext getContext(Class<?> type)
{
   if (type.equals(WhateverClassIsOverridedFor.class)) return
JAXBContext.newInstance()...;
}
 }
```

@Produces アノテーションを提供してコンテキスト向けのメディアタイプを指定しなければなりません。 **ContextResolver<JAXBContext>** を実装する必要もあります。 これにより、 ランタイムが正しいコンテキストリゾルバを一致するようにします。 また、 **ContextResolver** クラスに **@Provider** アノテーションを付ける必要があります。

ContextResolver を使用できるようにする方法はいくつかあります。

1. **javax.ws.rs.core.Application** 実装よりクラスまたはインスタンスとして返します。
2. **resteasy.providers** でプロバイダとしてリストします。
3. **WAR** ファイル内で RESTEasy が自動的にスキャンするようにします (詳細は設定ガイドを参照)。
4. **ResteasyProviderFactory.getInstance().registerProvider(Class)** または **registerProviderInstance(Object)** より手作業で追加します。

17.3. JAXB および XML プロバイダ

RESTEasy は XML に必要な JAXB プロバイダサポートを提供します。 アプリケーションのコーディングを簡単にする追加のアノテーションが複数あります。

17.3.1. @XmlHeader および @Stylesheet

XML 文書を出力する時に XML ヘッダを設定するには、**@org.jboss.resteasy.annotations.providers.jaxb.XmlHeader** アノテーションを使用します。

```
@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;
   }
}
```

ここでは、 **@XmlHeader** が XML 出力上で **xml-stylesheet** ヘッダを強制します。 **Thing** クラスにヘッダを置くと同じ結果を得ることができます。 RESTEasy によって提供される代替値については JavaDoc を参照してください。

RESTEasy にはスタイルシートのヘッダに対する便利なアノテーションもあります。 例は次の通りです。

```
@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;
    }
}
```

17.4. JAXB および JSON プロバイダ

RESTEasy では、 Jettison JSON ライブラリで JAXB アノテーションが付けられた POJO を JSON 間でマーシャリングすることができます。 Jettison の詳細は http://jettison.codehaus.org/ を参照してください。

Jettison には、 デフォルトの **Jettison Mapped Convention** 形式と BadgerFish の 2 つのマッピング形式が用意されています。

次の JAXB クラスを例とします。

```
@XmlRootElement(name = "book")
public class Book {

    private String author;
    private String ISBN;
    private String title;

    public Book() {
    }

    public Book(String author, String ISBN, String title) {
        this.author = author;
        this.ISBN = ISBN;
        this.title = title;
    }

    @XmlElement
    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
```

```
    }

    @XmlElement
    public String getISBN() {
       return ISBN;
    }

    public void setISBN(String ISBN) {
       this.ISBN = ISBN;
    }

    @XmlAttribute
    public String getTitle() {
       return title;
    }

    public void setTitle(String title) {
       this.title = title;
    }
}
```

BadgerFish の規則を使用して JAXB **Book** クラスが JSON へマーシャリングされます。

```
{"book":
   {
      "@title":"EJB 3.0",
      "author":{"$":"Bill Burke"},
      "ISBN":{"$":"596529260"}
   }
}
```

エレメント値はマップに関連付けられています。 エレメントの値を見つけるには、 **$** 変数にアクセスする必要があります。 このような book は JavaScript でアクセスできます。

```
var data = eval("(" + xhr.responseText + ")");
document.getElementById("zone").innerHTML = data.book.@title;
document.getElementById("zone").innerHTML += data.book.author.$;
```

BadgerFish の規則を使用するには、 マーシャリングまたはアンマーシャリングする JAXB クラスか、 JAX-RS リソースメソッドまたはパラメータ上で **@org.jboss.resteasy.annotations.providers.jaxb.json.BadgerFish** アノテーションを使用する必要があります。

```
@BadgerFish
@XmlRootElement(name = "book")
public class Book {...}
```

JAXB クラスを RESTEasy アノテーションで汚染せずに JAX-RS メソッド上で **book** を返すには、 JAX-RS メソッドへアノテーションを追加します。

```
@BadgerFish
@GET
public Book getBook(...) {...}
```

入力が **book** である場合、 パラメータ上に配置します。

```
@POST
public void newBook(@BadgerFish Book book) {...}
```

デフォルトの Jettison でマップされた規則は次の JSON を返します。

```
{ "book" :
     {
        "@title":"EJB 3.0",
        "author":"Bill Burke",
        "ISBN":596529260
      }
}
```

title の前に @ 文字が付いていることに注意してください。 BadgerFish の規則とは異なり、 エレメントテキストの値を表すものではないため、 より簡単です (実用的なデフォルトです)。 次のように JavaScript でアクセスします。

```
var data = eval("(" + xhr.responseText + ")");
document.getElementById("zone").innerHTML = data.book.@title;
document.getElementById("zone").innerHTML += data.book.author;
```

マップされた規則により、
@org.jboss.resteasy.annotations.providers.jaxb.json.Mapped アノテーションを用いて JAXB マッピングを調整できるようになります。 これにより、 XML の名前空間を JSON の名前空間マッピングへ提供することができます。 例えば、 **package-info.java** クラス内に JAXB 名前空間を定義するには、 次のようになります。

```
@javax.xml.bind.annotation.XmlSchema(namespace="http://jboss.org/books")
package org.jboss.resteasy.test.books;
```

JSON から XML への名前空間マッピングを定義しないと例外が発生します。

```
java.lang.IllegalStateException: Invalid JSON namespace:
  http://jboss.org/books
at org.codehaus.jettison.mapped.MappedNamespaceConvention
  .getJSONNamespace(MappedNamespaceConvention.java:151)
at org.codehaus.jettison.mapped.MappedNamespaceConvention
  .createKey(MappedNamespaceConvention.java:158)
at org.codehaus.jettison.mapped.MappedXMLStreamWriter
  .writeStartElement(MappedXMLStreamWriter.java:241)
```

@Mapped アノテーションはこの問題を修正します。 JAXB クラス、 JAX-RS リソースメソッド、 アンマーシャリングするパラメータのいずれかに **@Mapped** アノテーションを付けます。

```
import org.jboss.resteasy.annotations.providers.jaxb.json.Mapped;
import org.jboss.resteasy.annotations.providers.jaxb.json.XmlNsMap;
```

```
 ...

@GET
@Produces("application/json")
@Mapped(namespaceMap = {
        @XmlNsMap(namespace = "http://jboss.org/books", jsonName =
"books")
})
public Book get() {...}
```

@XmlAttribute が **XMLElements** としてマーシャリングされるよう強制することもできます。

```
            @Mapped(attributeAsElements={"title"})
            @XmlRootElement(name = "book")
            public class Book {...}
```

JAXB クラスを RESTEasy アノテーションで汚染せずに JAX-RS メソッド上で **book** を返すには、JAX-RS メソッドへアノテーションを追加します。

```
            @Mapped(attributeAsElements={"title"})
            @GET
            public Book getBook(...) {...}
```

入力が **book** である場合、 パラメータ上に配置します。

```
 @POST
 public void newBook(@Mapped(attributeAsElements={"title"}) Book book)
{...}
```

17.5. JAXB および FASTINFOSET プロバイダ

RESTEasy は JAXB アノテーションが付けられたクラスを使用して **Fastinfoset** MIME タイプをサポートします。 **Fastinfoset** 文書は論理的に同等の XML 文書よりも迅速にシリアライズや解析ができ、 大きさも小さいため、 XML 文書のサイズや処理時間に問題がある場合に使用することができます。 XML JAXB プロバイダと同じように設定することができます。

17.6. JAXB オブジェクトのアレイとコレクション

RESTEasy は **java.util.Set** や JAXB オブジェクトの **java.util.List** のアレイを XML、JSON、 **Fastinfoset**、 その他の RESTEasy JAXB マッパ間で自動的にマーシャリングします。

```
 @XmlRootElement(name = "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("/")
 public class MyResource
 {
   @PUT
   @Path("array")
   @Consumes("application/xml")
   public void putCustomers(Customer[] customers)
   {
      Assert.assertEquals("bill", customers[0].getName());
      Assert.assertEquals("monica", customers[1].getName());
   }

   @GET
   @Path("set")
   @Produces("application/xml")
   public Set<Customer> getCustomerSet()
   {
      HashSet<Customer> set = new HashSet<Customer>();
      set.add(new Customer("bill"));
      set.add(new Customer("monica"));

      return set;
   }


   @PUT
   @Path("list")
   @Consumes("application/xml")
   public void putCustomers(List<Customer> customers)
   {
      Assert.assertEquals("bill", customers.get(0).getName());
      Assert.assertEquals("monica", customers.get(1).getName());
   }
 }
```

前述のリソースは、 JAXB オブジェクトを公開し、 受け取ります。 リソースは次のようなコレクションエレメントにラッピングされることとします。

```
 <collection>
<customer><name>bill</name></customer>
<customer><name>monica</name></customer>
 <collection>
```

パラメータかメソッド上で **@org.jboss.resteasy.annotations.providers.jaxb.Wrapped** アノテーションを使用すると、名前空間 URL や名前空間タグ、 コレクションエレメント名を変更することができます。

```
 @Target({ElementType.PARAMETER, ElementType.METHOD})
 @Retention(RetentionPolicy.RUNTIME)
 public @interface Wrapped
 {
String element() default "collection";

String namespace() default "http://jboss.org/resteasy";

String prefix() default "resteasy";
 }
```

次の XML を出力したいとしましょう。

```
 <foo:list xmlns:foo="http://foo.org">
<customer><name>bill</name></customer>
<customer><name>monica</name></customer>
 </foo:list>
```

この場合、 次のように **@Wrapped** アノテーションを使用します。

```
   @GET
   @Path("list")
   @Produces("application/xml")
   @Wrapped(element="list", namespace="http://foo.org", prefix="foo")
   public List<Customer> getCustomerSet()
   {
      List<Customer> list = new ArrayList<Customer>();
      list.add(new Customer("bill"));
      list.add(new Customer("monica"));

      return list;
   }
```

17.6.1. JSON および JAXB コレクション / アレイ

RESTEasy は JSON とのコレクションの使用をサポートします。 簡単な JSON アレイ内の返された JAXB オブジェクトのリストやセット、 アレイが含まれます。 次の例を見てみましょう。

```
   @XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Foo
{
   @XmlAttribute
   private String test;

   public Foo()
   {
   }
```

```
    public Foo(String test)
    {
       this.test = test;
    }

    public String getTest()
    {
       return test;
    }

    public void setTest(String test)
    {
       this.test = test;
    }
 }
```

次のように **Foo** クラスのリストやアレイが JSON で示されます。

```
[{"foo":{"@test":"bill"}},{"foo":{"@test":"monica}"}}]
```

出力を受け取る時もこの形式が必要です。

17.7. JAXB オブジェクトのマップ

ESTEasy は JAXB オブジェクトのマップを XML、 JSON、 **Fastinfoset**、 その他の RESTEasy JAXB マッパ間で自動的にマーシャリングします。 パラメータやメソッドの戻りタイプは、 ストリングをキーや JAXB オブジェクトとしたジェネリックタイプでなければなりません。

```
@XmlRootElement(namespace = "http://foo.com")
public static class Foo
{
   @XmlAttribute
   private String name;

   public Foo()
   {
   }

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

   public String getName()
   {
      return name;
   }
}

@Path("/map")
public static class MyResource
{
   @POST
   @Produces("application/xml")
   @Consumes("application/xml")
```

```
   public Map<String, Foo> post(Map<String, Foo> map)
   {
      Assert.assertEquals(2, map.size());
      Assert.assertNotNull(map.get("bill"));
      Assert.assertNotNull(map.get("monica"));
      Assert.assertEquals(map.get("bill").getName(), "bill");
      Assert.assertEquals(map.get("monica").getName(), "monica");
      return map;
   }
 }
```

このリソースは、 マップ内で JAXB オブジェクトを公開し、 受け取ります。 デフォルトでは、 デフォルトの名前空間内の **map** エレメントにラッピングされます。 各 **map** エレメントはゼロ以上の **entry** エレメントと 1 つの **key** 属性を持っています。

```
 <map>
<entry key="bill" xmlns="http://foo.com">
    <foo name="bill"/>
</entry>
<entry key="monica" xmlns="http://foo.com">
    <foo name="monica"/>
</entry>
 </map>
```

パラメータかメソッド上で **@org.jboss.resteasy.annotations.providers.jaxb.WrappedMap** アノテーションを使用すると、 名前空間 URL、 名前空間プレフィックスとマップ、 エントリ、 キーエレメント、 属性名を変更することができます。

```
 @Target({ElementType.PARAMETER, ElementType.METHOD})
 @Retention(RetentionPolicy.RUNTIME)
 public @interface WrappedMap
 {
/**
 * map element name
 */
String map() default "map";

/**
 * entry element name *
 */
String entry() default "entry";

/**
 * entry's key attribute name
 */
String key() default "key";

String namespace() default "";

String prefix() default "";
 }
```

次の XML を出力したいとします。

```
 <hashmap>
<hashentry hashkey="bill" xmlns:foo="http://foo.com">
    <foo:foo name="bill"/>
</hashentry>
 </map>
```

この場合、次のように **@WrappedMap** アノテーションを使用します。

```
@Path("/map")
public static class MyResource
{
   @GET
   @Produces("application/xml")
   @WrappedMap(map="hashmap", entry="hashentry", key="hashkey")
   public Map<String, Foo> get()
   {
      ...
      return map;
   }
```

17.7.1. JSON および JAXB マップ

RESTEasy は JSON とのマップの使用をサポートします。 簡単な JSON マップ内の返された JAXB オブジェクトが含まれます。 次の例を見てみましょう。

```
   @XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public static class Foo
{
   @XmlAttribute
   private String test;

   public Foo()
   {
   }

   public Foo(String test)
   {
      this.test = test;
   }

   public String getTest()
   {
      return test;
   }

   public void setTest(String test)
   {
      this.test = test;
   }
}
```

次のように **Foo** クラスのリストやアレイが JSON で示されます。

```
 { "entry1" : {"foo":{"@test":"bill"}}, "entry2" : {"foo":
{"@test":"monica}"}}}
```

出力にもこの形式が必要です。

17.7.2. Jettison プロバイダの考えられる問題

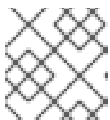
resteasy-jackson-provider-xxx.jar がクラスパスにあると、 Jackson JSON プロバイダがトリガされますが、 Jettison JAXB や JSON プロバイダに依存するコードでは問題となることがあります。これを修正するには、 **WEB-INF/lib** かクラスパスから Jackson を削除するか、 JAXB クラスに **@NoJackson** アノテーションを使用します。

17.8. インターフェース、 抽象クラス、 JAXB

オブジェクトモデルによっては抽象クラスやインターフェースを多く使用するものがあります。 JAXB はルートエレメントのインターフェースでは動作しません。 **JAXBContext** の作成に必要な情報が欠けているため、 RESTEasy はインターフェースや raw 抽象クラスであるパラメータをアンマーシャリングすることはできません。 次の例を見てください。

```
 public interface IFoo {}

 @XmlRootElement
 public class RealFoo implements IFoo {}

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

@PUT
@Consumes("application/xml")
public void put(IFoo foo) {...}
 }
```

この例では、 RESTEasy はエラーを表示します ("Cannot find MessageBodyReader for..." など)。 これは、 **IFoo** の実装が JAXB クラスであることを RESTEasy が認識しないため、 **IFoo** の **JAXBContext** を作成できないからです。 回避策として、 インターフェースに **@XmlSeeAlso** アノテーションを付けて問題を修正します。



注記

この方法は手動のハンドコード JAXB では機能しません。

```
 @XmlSeeAlso(RealFoo.class)
 public interface IFoo {}
```

IFoo 上にある追加の **@XmlSeeAlso** により、 **RealFoo** インスタンスのアンマーシャリング方法を認識する **JAXBContext** を RESTEasy が作成できるようになります。

第18章 RESTEASY ATOM のサポート

Atom は **フィード** と呼ばれる関連情報の一覧をコンパイルする XML ベースの書式です。 フィードは **エントリ** と呼ばれる複数の項目によって構成され、 各エントリには拡張可能なメタデータセット (タイトルなど) が含まれます。

Atom は主に Web コンテンツ (ウェブブログやニュースの見出しなど) を Web サイトにシンジケート化し、 直接ユーザーエージェントへシンジケート化します。

Atom は次世代の RSS フィードです。 主にウェブブログやニュースのシンジケート化に使用されますが、 フォーマットが分散通知やジョブキューなど Web サービスのエンベロープとして使用されたり、サービスへバルクデータを送受信するために使用されつつあります。

18.1. RESTEASY ATOM API とプロバイダ

RESTEasy は簡単なオブジェクトモデルを定義して Java で Atom を示し、 JAXB を使用してマーシャリングおよびアンマーシャリングを行います。 **org.jboss.resteasy.plugins.providers.atom** パッケージに主なクラスである **Feed**、 **Entry**、 **Content**、 **Link** が含まれています。 各クラスは JAXB アノテーションが付けられます。 ディストリビューションにはモデルを理解するのに大変便利な本プロジェクトの JavaDocs も含まれています。 以下のコードは、 RESTEasy API で Atom フィードを送信する簡単な例です。

```
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("Bill Burke"));
      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;
    }
}
```

RESTEasy の Atom プロバイダは JAXB ベースであるため、 Atom オブジェクトの XML による送信のみに限定されません。 RESTEasy の別の JAXB プロバイダ (JSON や FastinfoSet) は自動的に再使用することができます。 必要なのは、 メインのサブタイプである **`@Produces("application/atom+json")`** か **`@Consumes("application/atom+fastinfoset")`** の前に **`+atom`** を追加することのみです 。

18.2. ATOM プロバイダで JAXB を使用

`org.jboss.resteasy.plugins.providers.atom.Content` クラスは、 エントリのコンテンツボディを形成する、 JAXB アノテーションが付けられたオブジェクトのマーシャリングやアンマーシャリングを実行できるようにします。 次のコードは、 エントリのコンテンツボディとして添付される **`Customer`** オブジェクトで **`Entry`** を送信する例です。

```
@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() メソッドは、 マーシャリングされる Java JAXB オブジェクトが返されていることをコンテンツオブジェクトに伝えます。 XML 以外の基本形式を使用する場合 (例えば **application/atom+json** など)、 添付される JAXB オブジェクトはその形式にマーシャリングされます。

入力が Atom ドキュメントである場合、 **Content.getJAXBObject(Class clazz)** を使用して **Content** より JAXB オブジェクトを抽出することもできます。 次のコードは **Content** より **Customer** オブジェクトを抽出する例です。

```
@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);
   }
}
```

18.3. APACHE ABDERA からの ATOM サポート

RESTEasy は Atom プロトコルとデータ形式の実装である Apache Abdera をサポートします。 Abdera は Apache web site を参照してください。

Abdera は本格的な Atom サーバーですが、 Abdera の **Feed** および **Entry** インターフェースタイプ間で Atom データ形式をマーシャリングおよびアンマーシャリングするための JAX-RS の統合のみを RESTEasy はサポートします。

18.3.1. Abdera と Maven

Abdera プロバイダは RESTEasy ディストリビューションには含まれていません。 WAR アーカイブの **pom** ファイルに Abdera プロバイダが含まれるようにするには、 以下を追加します。 必ずこのコードにあるバージョンを実際使用している RESTEasy のバージョンに変更するようにしてください。

警告

RESTEasy は最新バージョンの Abdera を選択しないことがあります。

```
<repository>
      <id>jboss</id>
      <url>http://repository.jboss.org/maven2</url>
   </repository>

      ...
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
```

```
      <artifactId>abdera-atom-provider</artifactId>
      <version>...version...</version>
   </dependency>
```

18.3.2. Abdera プロバイダの使用

```
import org.apache.abdera.Abdera;
import org.apache.abdera.factory.Factory;
import org.apache.abdera.model.Entry;
import org.apache.abdera.model.Feed;
import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.methods.GetMethod;
import org.apache.commons.httpclient.methods.PutMethod;
import org.apache.commons.httpclient.methods.StringRequestEntity;
import org.jboss.resteasy.plugins.providers.atom.AbderaEntryProvider;
import org.jboss.resteasy.plugins.providers.atom.AbderaFeedProvider;
import org.jboss.resteasy.test.BaseResourceTest;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriInfo;
import javax.xml.bind.JAXBContext;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.Date;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public class AbderaTest extends BaseResourceTest
{

   @Path("atom")
   public static class MyResource
   {
      private static final Abdera abdera = new Abdera();

      @GET
      @Path("feed")
      @Produces(MediaType.APPLICATION_ATOM_XML)
      public Feed getFeed(@Context UriInfo uri) throws Exception
      {
         Factory factory = abdera.getFactory();
         Assert.assertNotNull(factory);
         Feed feed = abdera.getFactory().newFeed();
         feed.setId("tag:example.org,2007:/foo");
```

```
        feed.setTitle("Test Feed");
        feed.setSubtitle("Feed subtitle");
        feed.setUpdated(new Date());
        feed.addAuthor("James Snell");
        feed.addLink("http://example.com");


        Entry entry = feed.addEntry();
        entry.setId("tag:example.org,2007:/foo/entries/1");
        entry.setTitle("Entry title");
        entry.setUpdated(new Date());
        entry.setPublished(new Date());
        entry.addLink(uri.getRequestUri().toString());

        Customer cust = new Customer("bill");

        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        StringWriter writer = new StringWriter();
        ctx.createMarshaller().marshal(cust, writer);
        entry.setContent(writer.toString(), "application/xml");
        return feed;

     }

     @PUT
     @Path("feed")
     @Consumes(MediaType.APPLICATION_ATOM_XML)
     public void putFeed(Feed feed) throws Exception
     {
        String content = feed.getEntries().get(0).getContent();
        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        Customer cust = (Customer) ctx.createUnmarshaller().unmarshal(new
StringReader(content));
        Assert.assertEquals("bill", cust.getName());

     }

     @GET
     @Path("entry")
     @Produces(MediaType.APPLICATION_ATOM_XML)
     public Entry getEntry(@Context UriInfo uri) throws Exception
     {
        Entry entry = abdera.getFactory().newEntry();
        entry.setId("tag:example.org,2007:/foo/entries/1");
        entry.setTitle("Entry title");
        entry.setUpdated(new Date());
        entry.setPublished(new Date());
        entry.addLink(uri.getRequestUri().toString());

        Customer cust = new Customer("bill");

        JAXBContext ctx = JAXBContext.newInstance(Customer.class);
        StringWriter writer = new StringWriter();
        ctx.createMarshaller().marshal(cust, writer);
        entry.setContent(writer.toString(), "application/xml");
        return entry;
```

```
      }

      @PUT
      @Path("entry")
      @Consumes(MediaType.APPLICATION_ATOM_XML)
      public void putFeed(Entry entry) throws Exception
      {
         String content = entry.getContent();
         JAXBContext ctx = JAXBContext.newInstance(Customer.class);
         Customer cust = (Customer) ctx.createUnmarshaller().unmarshal(new
StringReader(content));
         Assert.assertEquals("bill", cust.getName());

      }
   }

   @Before
   public void setUp() throws Exception
   {

dispatcher.getProviderFactory().registerProvider(AbderaFeedProvider.class)
;

dispatcher.getProviderFactory().registerProvider(AbderaEntryProvider.class
);
      dispatcher.getRegistry().addPerRequestResource(MyResource.class);
   }

   @Test
   public void testAbderaFeed() throws Exception
   {
      HttpClient client = new HttpClient();
      GetMethod method = new GetMethod("http://localhost:8081/atom/feed");
      int status = client.executeMethod(method);
      Assert.assertEquals(200, status);
      String str = method.getResponseBodyAsString();

      PutMethod put = new PutMethod("http://localhost:8081/atom/feed");
      put.setRequestEntity(new StringRequestEntity(str,
MediaType.APPLICATION_ATOM_XML, null));
      status = client.executeMethod(put);
      Assert.assertEquals(200, status);

   }

   @Test
   public void testAbderaEntry() throws Exception
   {
      HttpClient client = new HttpClient();
      GetMethod method = new
GetMethod("http://localhost:8081/atom/entry");
      int status = client.executeMethod(method);
      Assert.assertEquals(200, status);
      String str = method.getResponseBodyAsString();
```

```
      PutMethod put = new PutMethod("http://localhost:8081/atom/entry");
      put.setRequestEntity(new StringRequestEntity(str,
MediaType.APPLICATION_ATOM_XML, null));
      status = client.executeMethod(put);
      Assert.assertEquals(200, status);
   }
}
```

第19章 JACKSON による JSON サポート

RESTEasy は、 JSON の Jettison JAXB アダプタを除く Jackson プロジェクトとの統合をサポートします。 Jackson の出力形式は BadgerFish や Jettison の形式よりも直感的に理解しやすいと思うユーザーが多く存在します。

Jackson は http://jackson.codehaus.org より入手可能です。 Jackson は JSON 間での Java オブジェクトのマーシャリングを容易にします 。Jackson には JavaBean ベースのモデルや JAXB と似た API があります。 RESTEasy は Jackson Tutorial の説明通りに JavaBean モデルと統合します。

ご自分のプロジェクトに Jackson を追加するには、次のようにビルドに Maven の依存関係を追加します。

```
<repository>
   <id>jboss</id>
   <url>http://repository.jboss.org/maven2</url>
</repository>

   ...
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-jackson-provider</artifactId>
   <version>1.1.GA</version>
</dependency>
```

RESTEasy は Jackson へビルドされた JAX-RS 統合を複数の方法で拡張します。 最初の拡張は **application/*+json** へのサポートを提供しました。 これまで、 Jackson は **application/json** と **text/json** のみを有効なメディアタイプとして許可しましたが、 **application/*+json** サポートにより Jackson で JSON ベースのメディアタイプをマーシャルできるようになりました。 例は次の通りです。

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

   @GET
   @Produces("application/vnd.customer+json")
   public Customer[] getCustomers() {}
}
```

RESTEasy の JAXB プロバイダを Jackson と使用すると問題が生じます。 Jackson を使用してJSON を出力するのではなく、 Jettison と JAXB を使用するようにしてください。 Jettison と JAXB を使用するには、 Jackson プロバイダをインストールしないか、 次のように JAXB アノテーションが付けられたクラス上で **@org.jboss.resteasy.annotations.providers.NoJackson** アノテーションを使用しないようにします。

```
@XmlRootElement
@NoJackson
public class Customer {...}

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

   @GET
```

```
        @Produces("application/vnd.customer+json")
        public Customer[] getCustomers() {}
    }
```

JAXB クラスに **@NoJackson** アノテーションが付けられない場合は、 メソッドパラメータにアノテーションを付けるようにします。

```
        @XmlRootElement
        public class Customer {...}

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

           @GET
           @Produces("application/vnd.customer+json")
           @NoJackson
           public Customer[] getCustomers() {}

           @POST
           @Consumes("application/vnd.customer+json")
           public void createCustomer(@NoJackson Customer[] customers)
{...}
        }
```

19.1. 考えられる JAXB プロバイダとの競合

Jackson クラスに JAXB アノテーションが付けられ、 クラスパスに **resteasy-jaxb-provider** がある場合、 Jettison JAXB マーシャリングコードをトリガすることができます。 JAXB JSON マーシャラを無効にするには、 クラスに **@org.jboss.resteasy.annotations.providers.jaxb.IgnoreMediaTypes("application/*+json")** アノテーションを付けます。

第20章 マルチパートプロバイダ

RESTEasy は **multipart/*** および **multipart/form-data** MIME (多目的インターネットメール拡張 : Multipurpose Internet Mail Extension) タイプをサポートします。 **multipart** MIME 形式はコンテンツボディのリストを渡します。 複数のコンテンツボディが 1 つのメッセージに組み込まれます。 **multipart/form-data** は多くの場合ウェブアプリケーションの HTML フォーム文書にあり、一般的にはファイルのアップロードに使用されます。 **form-data** 形式は、コンテンツのインライン化された各部分にそれに関連した名前がある以外は他の **multipart** 形式のように動作します。

RESTEasy は、 **multipart** タイプの読み書きや、任意のリスト (すべての **multipart** タイプ) や Map (**multipart/form-data** のみ) オブジェクトのマーシャルに対してカスタム API を提供します。

20.1. MULTIPART/MIXED による入力

JAX-RS サービスを書く時、 RESTEasy は **org.jboss.resteasy.plugins.providers.multipart.MultipartInput** インターフェースを提供し、 すべての **multipart** MIME タイプを読み取れるようにします。

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

public interface MultipartInput
{
   List<InputPart> getParts();

   String getPreamble();
}

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();
}
```

MultipartInput は、 **multipart** メッセージの各パートにアクセスできるようにする簡単なインターフェースです。 各パートは **InputPart** インターフェースによって表され、 ヘッダのセットと関連しています。 1 つのパートをアンマーシャリングするには **getBody()** メソッドの 1 つを呼び出します。 **Type genericType** パラメータは null でも問題ありませんが、 **Class type** パラメータには null 以外を設定しなければなりません。 RESTEasy はパートのメディアタイプとパスするタイプ情報を基に **MessageBodyReader** を検索します。 次のコードは、 XML のパートを **Customer** と呼ばれる JAXB アノテーションが付けられたクラスへアンマーシャリングします。

```
   @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);
         }
      }
   }
```

ジェネリックタイプのメタデータに反応するボディーパートをアンマーシャリングしたい場合は、 次のように **org.jboss.resteasy.util.GenericType** クラスを使用します。

```
   @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<<() {});
         }
      }
   }
```

GenericType のみがランタイム時にジェネリックタイプの情報を取得できるため、 **GenericType** が必要となります。

20.2. JAVA.UTIL.LIST とマルチパートデータ

ボディーパートが同一の場合、 **java.util.List** を入力パラメータとして提供すると、 各パートの手作業によるアンマーシャリングを省略することができます。 下記のコード例にあるように、 リストタイプ宣言のジェネリックパラメータでアンマーシャリングされるタイプが含まれるようにしなければなりません。

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

20.3. MULTIPART/FORM-DATA による入力

JAX-RS サービスを書く時、 RESTEasy はインターフェースを提供し、 **multipart/form-data** MIME タイプを読み取れるようにします。 **multipart/form-data** は多くの場合ウェブアプリケーションの HTML フォーム文書にあり、一般的にはファイルのアップロードに使用されます。 **form-data** 形式は、コンテンツのインライン化された各パートが名前と関連している以外は他の **multipart** 形式と同様に動作します。**form-data** 入力のインターフェースは **org.jboss.resteasy.plugins.providers.multipart.MultipartFormDataInput** になります。

```
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;
}
```

本章の最初に説明した **MultipartInput** と同様に動作します。

20.4. MULTIPART/FORM-DATA を使用した JAVA.UTIL.MAP

form-data を使用して、ボディーパートが同一の場合、 **java.util.Map** を入力パラメータとして提供すると、 手作業による各パートのアンマーシャリングを省略することができます。 下記のコード例にあるように、 リストタイプ宣言のジェネリックパラメータでアンマーシャリングされるタイプが含まれるようにしなければなりません。

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

20.5. MULTIPART/RELATED による入力

JAX-RS サービスを書く時、 RESTEasy はインターフェースを提供し、 **multipart/related** MIME タイプを読み取れるようにします。 **multipart/related** は、 メッセージパートを個別に扱うのではなく全体を 1 つとして扱うことを表しています。 **multipart/related** を使用すると、イメージが付いた Web ページを単一メッセージで送信するようなタスクを実行できます。

各 **multipart/related** メッセージには、 メッセージの他のパートを参照する **root/start** パートがあります。 パートは **Content-ID** ヘッダによって識別されます。 **multipart/related** は RFC 2387 によって定義されます。 **related** 入力のインターフェースは

org.jboss.resteasy.plugins.providers.multipart.MultipartRelatedInput です。

```
public interface MultipartRelatedInput extends MultipartInput
{
   String getType();

   String getStart();

   String getStartInfo();

   InputPart getRootPart();

   Map<String, InputPart> getRelatedMap();
}
```

本章の最初に説明した **MultipartInput** と同様に動作します。

20.6. マルチパートによる出力

RESTEasy は簡単な API を備えており、マルチパートデータを出力します。

```
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()
}
```

multipart データを出力するには、 **MultipartOutput** オブジェクトを作成し、 **addPart()** メソッ

ドを呼び出します。 RESTEasy は自動的に **MessageBodyWriter** を検索し、 エンティティオブジェクトをマーシャリングします。 **MultipartInput** 同様、 マーシャリングがジェネリックタイプのメタデータに影響を受けることがあります。 この場合、 **GenericType** を使用します。 次の例は、 呼び出しているクライアントに **multipart/mixed** 形式を返します。 **application/xml** へマーシャリングする、 JAXB アノテーションを付けられた **Customer** オブジェクトがパートとなります。

```
   @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;
      }
```

20.7. マルチパート出力と JAVA.UTIL.LIST

ボディーパートが同一の場合、 **java.util.Map** を入力パラメータとして提供すると、 各パートの手作業によるアンマーシャリングや **MultipartOutput** オブジェクトの使用を省略できます。 下記のコード例にあるように、 リストタイプ宣言のジェネリックパラメータでアンマーシャリングされるタイプが含まれるようにしなければなりません 。また、 メソッドに **@PartType** アノテーションを付けて各パートのメディアタイプを指定しなければなりません。 次の例では、カスタマリストをクライアントへ返し、各カスタマは JAXB オブジェクトです。

```
   @Path("/multipart")
   public class MyService
   {
      @GET
      @Produces("multipart/mixed")
      @PartType("application/xml")
      public List<Customer> get()
      {
         ...
      }
   }
```

20.8. MULTIPART/FORM-DATA による出力

RESTEasy は簡単な API を備えており、multipart/form-dat を出力します。

```
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()
}
```

multipart/form-data を出力するには、 **MultipartOutput** オブジェクトを作成し、 **addFormData()** メソッドを呼び出します。 RESTEasy は自動的に **MessageBodyWriter** を検索し、エンティティオブジェクトをマーシャリングします。 **MultipartInput** 同様、 マーシャリングがジェネリックタイプのメタデータに影響を受けることがあります。 この場合、 **GenericType** を使用します。 次の例は、 呼び出しているクライアントに **multipart/form-data** 形式を返します。 **application/xml** へマーシャリングする、 JAXB アノテーションが付けられた **Customer** オブジェクトがパートとなります。

```
   @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;
      }
```

20.9. JAVA.UTIL.MAP を使用したマルチパート FORMDATA 出力

ボディーパートが同一の場合、 **java.util.Map** を入力パラメータとして提供すると、 各パートの手作業によるアンマーシャリングや **MultipartFormDataOutput** オブジェクトの使用を省略できます。 下記のコード例にあるように、 リストタイプ宣言のジェネリックパラメータでアンマーシャリングされるタイプが含まれるようにしなければなりません 。また、 メソッドに **@PartType** アノテーションを付けて各パートのメディアタイプを指定しなければなりません。 次の例では、カスタマリストをクライアントへ返し、各カスタマは JAXB オブジェクトです。

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

```
      }
   }
```

20.10. MULTIPART/RELATED による出力

RESTEasy は簡単な API を備えており、**multipart/related** を出力します。

```
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)
}
```

multipart/related を出力するには、 **MultipartRelatedOutput** オブジェクトを作成し、 **addPart()** メソッドを呼び出します。 最初に追加されたパートが **multipart/related** メッセージのルートパートとして使用されます。 RESTEasy は自動的に **MessageBodyWriter** を検索し、 エンティティオブジェクトをマーシャリングします。 **MultipartInput** 同様、 マーシャリングがジェネリックタイプのメタデータに影響を受けることがあります。 この場合、 **GenericType** を使用します。 次の例は、 呼び出しているクライアントに **multipart/related** 形式 (2 つのイメージがある HTML ファイル) を返します。

```
@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>
"
            + "This is me: <img src='cid:http://example.org/me.png' />
"
            + "<br />This is you: <img
src='cid:http://example.org/you.png' />
"
            + "</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;
     }
  }
```

20.11. @MULTIPARTFORM と POJO

multipart/form-data パケットを理解している場合は、 **@org.jboss.resteasy.annotations.providers.multipart.MultipartForm** アノテーションと **@FormParam** JAX-RS アノテーションを使用して **multipart/form-data** パケットを POJO クラス間でマップすることができます。 マップするには、 最低でもデフォルトのコンストラクタで POJO を定義し、 フィールドとプロパティのいずれかか両方に **@FormParam** アノテーションを付けます。 **@FormParam** を出力するには、 **@org.jboss.resteasy.annotations.providers.multipart.PartType** アノテーションが付けられていなければなりません。 例は次の通りです。

```
     public class CustomerProblemForm {
        @FormData("customer")
        @PartType("application/xml")
        private Customer customer;

        @FormData("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; }
     }
```

POJO クラスを定義したら、 POJO クラスを使用して **multipart/form-data** を表すことができます。 次のコードでは RESTEasy クライアントフレームワークを使用して **CustomerProblemForm** を送ります。

```
     @Path("portal")
     public interface CustomerPortal {

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

     {
```

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

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

@MultipartForm アノテーションは、 オブジェクトに **@FormParam** があるため、 パラメータよりマーシャリングされるべきであることを RESTEasy に伝えます。 同じオブジェクトを使用して **multipart** データを受信することができます。 カスタマポータルのサーバー側の例は次の通りです。

```
    @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...
       }
    }
```

20.12. XOP (XML バイナリ最適パッケージ化)

RESTEasy は、 XOP (XML バイナリ最適パッケージ化: XML-binary Optimized Packaging) メッセージの **multipart/related** としてのパッケージ化をサポートします。 そのため、 一部バイナリコンテンツを持つ JAXB アノテーションが付けられた POJO がある場合、 他の方法でバイナリをエンコードしなくても送信できます。 これにより、 POJO の便利性を維持しながらトランスポートを迅速にします (XOP の詳細は W3C web page を参照してください)。

次の JAXB アノテーションが付けられた POJO を例として見てみましょう。 **@XmlMimeType** は JAXB にバイナリコンテンツの MIME タイプを伝えます (これは必須ではありませんが推奨されます)。

```
@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 ...
}
```

この例では、 **myBinary** と **myDataHandler** がバイナリの添付として処理され、 XOP オブジェクトは XML として送信されます。 **javax.activation.DataHandler** が最も一般的なサポート対象タイ

プであるため、 **java.io.InputStream** か **javax.activation.DataSource** が必要な場合は **DataHandler** を使用する必要があります。 **java.awt.Image** と **javax.xml.transform.Source** もサポートされます。 前述の例で **Customer** が JAXB 対応の POJO であることを仮定すると、 前述の例を送信する Java クライアントの例は次のようになります。

```
// our client interface:
@Path("mime")
public static interface MultipartClient {
  @Path("xop")
  @PUT
  @Consumes(MediaType.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);
}
```

@Consumes(MediaType.MULTIPART_RELATED) は、 XOP メッセージを保持する形式である **multipart/related** のパッケージを送信したいことを RESTEasy に伝えます。 **@XopWithMultipartRelated** は、 XOP メッセージを作成したいことを RESTEasy に伝えます。 これで、 POJO とクライアントサービスが XOP メッセージを送信できるようになったため、 次に XOP メッセージを読めるサーバーが必要となります。

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

@Consumes(MediaType.MULTIPART_RELATED) は、 **multipart/related** パッケージを読みたいことを RESTEasy に伝えます。 **@XopWithMultipartRelated** は、 XOP メッセージを読みたいことを RESTEasy に伝えます。XOP の戻り値を作成するには **@Produce** でそれらにアノテーションを付けます。

第21章 YAML プロバイダ

Beta 6 より RESTEasy に JYML ライブラリを持つ YAML の内蔵サポートが含まれるようになりました。 YAML のサポートを有効にするには、 RESTEasy のクラスパスに **jyaml-1.3.jar** を追加します。

JYAML **JAR** は SourceForge よりダウンロードできます。

Maven を使用する場合、JYAML **JAR** はメインレポジトリより使用でき、次の依存関係が含まれます。

```
 <dependency>
<groupId>org.jyaml</groupId>
<artifactId>jyaml</artifactId>
<version>1.3</version>
 </dependency>
```

RESTEasy を開始する時、 **YamlProvider** が追加されたことを示す行をログで確認してください。これは、 RESTEasy が JYAML **JAR** を見つけたことを意味します。

```
2877 Main INFO org.jboss.resteasy.plugins.providers.RegisterBuiltin -
Adding YamlProvider
```

YAML プロバイダは 3つの MIME タイプを認識します。

- **text/x-yaml**
- **text/yaml**
- **application/x-yaml**

次のようにリソースメソッドで YAML を使用することができます。

```
 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();
}
...
 }
```

第22章 ストリングベース @*PARAM のストリングマーシャリング

@PathParam、 **@QueryParam**、 **@MatrixParam**、 **@FormParam**、 **@HeaderParam** は raw HTTP 要求のストリングとして表されます。 **valueOf(String)** 静的メソッドがある場合や 1 つのストリングパラメータを取るコンストラクタがある場合、 これら挿入されたパラメータタイプをオブジェクトに変換することができます。 **valueOf()** を持つクラスがある場合や、 HTTP 要求に不適切なストリングコンストラクタがある場合、 RESTEasy のプロプラエタリ **@Provider** をプラグインできます。

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

public interface StringConverter<T>
{
   T fromString(String str);
   String toString(T value);
}
```

このインターフェースはカスタマイズした独自のストリングマーシャリングを使用できるようにします。 **resteasy.providers context-param** の **web.xml** に登録されます (詳細は「インストール/設定」章を参照してください)。 **ResteasyProviderFactory.addStringConverter()** メソッドを呼び出して手作業で登録できます。 次は **StringConverter** を使用した簡単な例です。

```
   import org.jboss.resteasy.client.ProxyFactory;
   import org.jboss.resteasy.spi.StringConverter;
   import org.jboss.resteasy.test.BaseResourceTest;
   import org.junit.Assert;
   import org.junit.Before;
   import org.junit.Test;

   import javax.ws.rs.HeaderParam;
   import javax.ws.rs.MatrixParam;
   import javax.ws.rs.PUT;
   import javax.ws.rs.Path;
   import javax.ws.rs.PathParam;
   import javax.ws.rs.QueryParam;
   import javax.ws.rs.ext.Provider;

   public class StringConverterTest extends BaseResourceTest
   {
      public static class POJO
      {
         private String name;

         public String getName()
         {
            return name;
         }

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

```
      @Provider
      public static class POJOConverter implements StringConverter<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();
         }
      }

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

      @Before
      public void setUp() throws Exception
      {

dispatcher.getProviderFactory().addStringConverter(POJOConverter.class);
         dispatcher.getRegistry().addPerRequestResource(MyResource.class);
      }

      @Path("/")
      public static interface MyClient
      {
         @Path("{pojo}")
         @PUT
         void put(@QueryParam("pojo")POJO q, @PathParam("pojo")POJO pp,
                  @MatrixParam("pojo")POJO mp, @HeaderParam("pojo")POJO
hp);
      }

      @Test
      public void testIt() throws Exception
      {
         MyClient client = ProxyFactory.create(MyClient.class,
```

```
"http://localhost:8081");
         POJO pojo = new POJO();
         pojo.setName("pojo");
         client.put(pojo, pojo, pojo, pojo);
      }
   }
```

第23章 JAVAX.WS.RS.CORE.RESPONSE を使用した応答

`javax.ws.rs.core.Response` クラスと `ResponseBuilder` クラスを使用してカスタム応答を作成することができます。 独自のストリーミングを実行するには、 エンティティの応答は `javax.ws.rs.core.StreamingOutput` の実装でなければなりません。 詳細は Java ドキュメントを参考してください。

第24章 例外処理

24.1. 例外マッパ

ExceptionMapper は、 アプリケーション例外をキャッチし、 特定の HTTP 応答を書くことができるカスタムのアプリケーション提供によるコンポーネントです。 **@Provider** と関連し、 次のインターフェースを実装します。

```
package javax.ws.rs.ext;

import javax.ws.rs.core.Response;

/**
* Contract for a provider that maps Java exceptions to
* {@link javax.ws.rs.core.Response}. An implementation of this
interface must
* be annotated with {@link Provider}.
*
* @see Provider
* @see javax.ws.rs.core.Response
*/
public interface ExceptionMapper<E>
{

/**
* Map an exception to a {@link javax.ws.rs.core.Response}.
*
* @param exception the exception to map to a response
* @return a response mapped from the supplied exception
*/
Response toResponse(E exception);
}
```

アプリケーションが例外を送出すると、 例外は JAX-RS ランタイムによってキャッチされます。 JAX-RS は登録された **ExceptionMapper** をスキャンし、送出された例外タイプのマーシャリングをサポートするものを検索します。 以下は **ExceptionMapper** の例になります。

```
@Provider
public class EJBExceptionMapper implements
ExceptionMapper<javax.ejb.EJBException>
{

Response toResponse(EJBException exception) {
return Response.status(500).build();
}

}
```

ExceptionMapper は **MessageBodyReader** や **MessageBodyWriter** と同様に登録されます。 RESTEasy プロバイダ **context-param** よりスキャンするか (**WAR** ファイルにデプロイする場合) **ResteasyProviderFactory** クラスよりプログラムにて実行します。

24.2. RESTEASY 内蔵の内部で送出された例外

RESTEasy にはディスパッチングやマーシャリング中にエラーが発生した時に送出された RESTEasy 内蔵の例外があります。 各例外は特定の HTTP エラーコードに一致します。全一覧は、**`org.jboss.resteasy.spi`** パッケージ以下にある RESTEasy Java ドキュメントを参照してください。次の表は最も一般的な例外を一覧表示しています。

表24.1 一般的な例外

例外	HTTP コード	説明
`BadRequestException`	400	不適切な要求です。要求が正しくフォーマットされていなかったか、要求入力の処理に問題があります。
`UnauthorizedException`	401	承認されていません。 RESTEasy の簡単なアノテーションベースおよびロールベースセキュリティを使用するとセキュリティ例外が送出されます。
`InternalServerErrorException`	500	内部のサーバーエラーです。
`MethodNotAllowedException`	405	メソッドが許可されていません。呼び出された HTTP 操作を処理できるリソースの JAX-RS メソッドはありません。
`NotAcceptableException`	406	許可できません。 Accept ヘッダにリストされているメディアタイプを生成できる JAX-RS メソッドがありません。
`NotFoundException`	404	見つかりません。 要求パス / リソースに対応する JAX-RS メソッドがありません。
`Failure`	該当なし	内部の RESTEasy です。 ログに記録されません。
`LoggableFailure`	該当なし	内部の RESTEasy エラーです。ログに記録されます。
`DefaultOptionsMethodException`	該当なし	ユーザーが JAX-RS メソッドなしで HTTP OPTIONS を呼び出すと、 RESTEasy はこの例外を送出してデフォルトの動作を提供します。

24.3. RESTEASY 内蔵の例外をオーバーライドする

RESTEasy 内蔵の例外をオーバーライドするには、 例外に対して **`ExceptionMapper`** を書きます。 また、 **`WebApplicationException`** など送出されるすべての例外に対して **`ExceptionMapper`** を書くことができます。

第25章 個別の JAX-RS リソース BEAN の設定

JAX-RS アノテーションが付けられたリソース Bean のパスをスキャンする時、 Bean は **リクエスト毎モード** に登録されます。 そのため、 HTTP 要求ごとに 1 つのインスタンスが作成されます。 通常、使用している環境からの情報が必要となります。Bata 2 やそれ以前の Servlet コンテナで **WAR** を実行する場合、 Java EE リソースや設定情報への参照取得に JNDI ルックアップのみしか使用できません。 この場合、 RESTEasy **WAR** ファイルの **web.xml** に EE 設定 (**ejb-ref**、 **env-entry**、 **persistence-context-ref** など) を定義し、 コード内で **java:comp** 名前空間にて JNDI ルックアップを実行します。 例は次の通りです。

web.xml

```
<ejb-ref>
  <ejb-ref-name>ejb/foo</ejb-ref-name>
  ...
</ejb-ref>
```

リソースコード:

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

   public Object getSomethingFromJndi() {
      new InitialContext.lookup("java:comp/ejb/foo");
   }
...
}
```

レジストリより手作業で Bean を設定したり登録することもできます。 **WAR** ベースのデプロイメントでは、そのために固有の **ServletContextListener** を書く必要があります。 次のように、 リスナによってレジストリへの参照の取得が可能になります。

```
public class MyManualConfig implements ServletContextListener
{
   public void contextInitialized(ServletContextEvent event)
   {

      Registry registry = (Registry)
event.getServletContext().getAttribute(Registry.class.getName());

   }
...
}
```

この処理を完全に理解するために、 Spring 統合と組み込みコンテナの Spring 統合を確認することが推奨されます。

第26章 GZIP 圧縮 / 展開

RESTEasy には自動 GZIP 展開のサポートがあります。 クライアントフレームワークや JAX-RS サービスが **gzip** の **Content-Encoding** でメッセージボディを受信すると、 メッセージは自動的に展開されます。 クライアントフレームワークは **gzip, deflate** への **Accept-Encoding** ヘッダも自動的に設定します。

RESTEasy は自動圧縮もサポートします。 **gzip** の **Content-Encoding** ヘッダを持つ要求または応答が送受信されると、 RESTEasy が圧縮を実行します。 各 **Content-Encoding** を手作業で設定したくない場合は **@org.jboss.resteasy.annotation.GZIP** アノテーションを使用することができます。

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

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

ここでは、 **order** メッセージボディが GZIP 圧縮のためタグ付けされています。 同じアノテーションを使用してサーバー応答をタグ付けすることができます。

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

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

第27章 RESTEASY キャッシング機能

RESTEasy は HTTP キャッシングセマンティクスをサポートするため多くのアノテーションを提供し、 Cache-Contro ヘッダ設定などのプロセスを簡略化したり、サーバー側とクライアント側の両方のインメモリキャッシュを使用できるようにします。

27.1. @CACHE と @NOCACHE アノテーション

RESTEasy は正常な GET 要求上に **Cache-Control** ヘッダを設定できるようにする JAX-RS への拡張を提供します。 **@GET** でアノテーション付けされたメソッドのみに使用できます。 正常な GET 要求は **200 OK** を応答として返します。

```
package org.jboss.resteasy.annotations.cache;

public @interface Cache
{
   int maxAge() default -1;
   int sMaxAge() default -1;
   boolean noStore() default false;
   boolean noTransform() default false;
   boolean mustRevalidate() default false;
   boolean proxyRevalidate() default false;
   boolean isPrivate() default false;
}

public @interface NoCache
{
   String[] fields() default {};
}
```

@Cache は複雑な **Cache-Control** ヘッダを構築します。 **@NoCache** は何もキャッシュしたくないことを指定します (**Cache-Control: nocache**)。

これらのアノテーションはリソースクラスやインターフェース、 個別の **@GET** リソースメソッド上に付けることができます。 これらのアノテーションは各 **@GET** リソースメソッドのデフォルトのキャッシュ値を指定します。

27.2. クライアント「ブラウザ」キャッシュ

RESTEasy では、 クライアントプロキシフレームワークや raw **ClientRequest** と使用するための、クライアント側のブラウザに似たキャッシュを作成することができます。 このキャッシュはサーバー応答と共に返される **Cache-Control** ヘッダの場所を見つけます。 クライアントが応答のキャッシュが可能であることを **Cache-Control** ヘッダが指定すると、 RESTEasy はローカルメモリ内に応答をキャッシュします。 このキャッシュは **max-age** 要件に従い、 **Last-Modified** ヘッダと **ETag** ヘッダの両方またはいずれかが元の応答と共に返されると、自動的に HTTP 1.1 キャッシュ再検証を実行します (**Cache-Control** やキャッシュ再検証に関する詳細は HTTP 1.1 仕様を参照してください)。

RESTEasy キャッシングは簡単に有効化することができます。 下記はクライアントプロキシフレームワークで使用されるクライアントキャッシュを表しています。

```
@Path("/orders")
public interface OrderServiceClient {
```

```
    @Path("{id}")
    @GET
    @Produces("application/xml")
    public Order getOrder(@PathParam("id") String id);
}
```

このインターフェースに対してプロキシを作成し、 次のようにプロキシのキャッシュを有効にすることができます。

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.client.cache.CacheFactory;
import org.jboss.resteasy.client.cache.LightweightBrowserCache;

public static void main(String[] args) throws Exception
{
      RegisterBuiltin.register(ResteasyProviderFactory.getInstance());
      OrderServiceClient proxy =
ProxyFactory.create(OrderServiceClient.class, generateBaseUrl());

      // This line enables caching
      LightweightBrowserCache cache = CacheFactory.makeCacheable(proxy);
}
```

呼び出しの実行にプロキシサーバーではなく **ClientRequest** クラスを使用している場合、 次のようにキャッシュを有効にすることができます。

```
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.client.cache.CacheFactory;
import org.jboss.resteasy.client.cache.LightweightBrowserCache;

public static void main(String[] args) throws Exception
{
      RegisterBuiltin.register(ResteasyProviderFactory.getInstance());

      // This line enables caching
      LightweightBrowserCache cache = new LightweightBrowserCache();

      ClientRequest request = new
ClientRequest("http://example.com/orders/333");
      CacheFactory.makeCacheable(request, cache);
}
```

デフォルトでは、 **LightweightBrowserCache** は 2 メガバイトの最大キャッシング領域を持っています。 これを変更するには、 プログラムで **setMaxBytes()** メソッドを呼び出します。 **キャッシュが満杯になるとキャッシュされたデータはすべて自動的に削除されます** 。 さらに複雑なキャッシングソリューションやサードパーティのキャッシュオプションをサポートするには、 resteasy-development リストへ連絡し、 コミュニティに相談してください。

27.3. ローカルのサーバー側応答キャッシュ

RESTEasy は、 JAX-RS サービスに対するローカルのサーバー側インメモリキャッシュを持っています。 JAX-RS リソースメソッドが **Cache-Control** ヘッダを設定する場合、 HTTP GET JAX-RS 呼び出しからマーシャリングされた応答を自動的にキャッシュします。 GET が受信されると、 URI が

キャッシュに保存されたかを RESTEasy サーバーキャッシュが確認します。 true の場合、 マーシャリングされた応答は JAX-RS メソッドを呼び出さずに返されます。 各キャッシュエントリには、 初期要求の **Cache-Control** ヘッダにある仕様が有効になる **最大期間** があります。 また、 キャッシュは応答ボディ上の MD5 ハッシュを使用して **ETag** も自動的に生成します。 これにより、 クライアントは **IF-NONE-MATCH** ヘッダで HTTP 1.1 キャッシュ再検証を実行できます。 最初のキャッシュヒットがない場合でもキャッシュは再検証を実行しますが、 JAX-RS メソッドは同じ **ETag** を持つボディを返します。

Maven でサーバー側のキャッシュを設定するには、 **resteasy-cache-core** アーティファクトを使用しなければなりません。

```
<dependency>
   <groupId>org.jboss.resteasy</groupId>
   <artifactId>resteasy-cache-core</artifactId>
   <version>1.1.GA</version>
</dependency>
```

次に、 **ServletContextListener** である **org.jboss.resteasy.plugins.cache.server.ServletServerCache** を追加します。 **web.xml** ファイルの **ResteasyBootstrap** リスナの後にこれを指定する必要があります。

```
<web-app>
    <listener>
        <listener-class>
            org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap
        </listener-class>
    </listener>

    <context-param>
        <param-name>resteasy.server.cache.maxsize</param-name>
        <param-value>1000</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.server.cache.eviction.wakeup.interval</param-
name>
        <param-value>5000</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.jboss.resteasy.plugins.cache.server.ServletServerCache
        </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>/rest-services/*</url-pattern>
    </servlet-mapping>

</web-app>
```

キャッシュの実装は JBoss Cache project を基にしています。 2 つの **context-param** 設定変数を設定することができます。 **resteasy.server.cache.maxsize** はキャッシュできるエレメント数を設定します。 **resteasy.server.cache.eviction.wakeup.interval** は、 陳腐化したエントリのキャッシュを削除するためバックグラウンドの排除スレッドを実行する割合を設定します。

第28章 インターセプタ

RESTEasy は JAX-RS 呼び出しを傍受し、**インターセプタ** と呼ばれるリスナと似たオブジェクトを介してルーティングすることができます。 サーバー側には 4 つの傍受ポイントがあります。

- **MessageBodyWriter** 呼び出しのラッピング
- **MessageBodyReader** 呼び出しのラッピング
- アンマーシャリングの前に受信要求を傍受する **プリプロセッサ** を経由
- JAX-RS メソッドの終了直後に呼び出される **ポストプロセッサ** を経由

MessageBodyReader や **MessageBodyWriter**、 クライアント側でのサーバーへのリモート呼び出しを傍受することもできます。

28.1. MESSAGEBODYREADER/WRITER インターセプタ

MessageBodyReader インターセプタと **MessageBodyWriter** インターセプタは、**MessageBodyReader.readFrom()** や **MessageBodyWriter.writeTo()** の呼び出しをラッピングします。 これらは **Output** や **InputStream** をラッピングするために使用されます。 例えば、 GZIP エンコーディングが動作するよう、 RESTEasy の GZIP サポートには **GzipOutputStream** や **GzipInputStream** を用いてデフォルトの **Output** と **InputStream** を作成しオーバーライドするインターセプタが含まれています。 応答 (クライアント側では送信要求) にヘッダを追加するためインターセプタを使用することもできます。

インターセプタを使用するには、**org.jbos.resteasy.spi.interception.MessageBodyReaderInterceptor** か **MessageBodyWriterInterceptor** を実装します。

```
public interface MessageBodyReaderInterceptor
{
   Object read(MessageBodyReaderContext context) throws IOException,
WebApplicationException;

}

public interface MessageBodyWriterInterceptor
{
   void write(MessageBodyWriterContext context) throws IOException,
WebApplicationException;

}
```

インターセプタは **MessageBodyWriterContext** や **MessageBodyReaderContext** によって操作されます。 インターセプタは Java コールスタックで一緒に呼び出されます。 後続のインターセプタを追加するには、 **MessageBodyReaderContext.proceed()** または **MessageBodyWriterContext.proceed()** を呼び出す必要があります。 呼び出すインターセプタがなくなったら、 **MessageBodyReader** または **MessageBodyWriter** の **readFrom()** か **writeTo()** メソッドを呼び出します。 このラッピングにより、 リーダーやライターに達する前にオブジェクトを変更でき、 **proceed()** が返信する時にクリーンアップを行うことができます。 **Context** オブジェクトは、 リーダーやライターに送信されたパラメータを変更するメソッドも持っています。

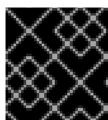
```
public interface MessageBodyReaderContext
```

```
{
   Class getType();

   void setType(Class type);

   Type getGenericType();

   void setGenericType(Type genericType);

   Annotation[] getAnnotations();

   void setAnnotations(Annotation[] annotations);

   MediaType getMediaType();

   void setMediaType(MediaType mediaType);

   MultivaluedMap<String, String> getHeaders();

   InputStream getInputStream();

   void setInputStream(InputStream is);

   Object proceed() throws IOException, WebApplicationException;
}

public interface MessageBodyWriterContext
{
   Object getEntity();

   void setEntity(Object entity);

   Class getType();

   void setType(Class type);

   Type getGenericType();

   void setGenericType(Type genericType);

   Annotation[] getAnnotations();

   void setAnnotations(Annotation[] annotations);

   MediaType getMediaType();

   void setMediaType(MediaType mediaType);

   MultivaluedMap<String, Object> getHeaders();

   OutputStream getOutputStream();

   public void setOutputStream(OutputStream os);

   void proceed() throws IOException, WebApplicationException;
}
```

MessageBodyReaderInterceptor と **MessageBodyWriterInterceptor** はサーバー側かクライアント側で使用することができます。 RESTeasy によって正しいインターセプタリストに追加されるよう、 **@org.jboss.resteasy.annotations.interception.ServerInterceptor** または **@org.jboss.resteasy.annotations.interception.ClientInterceptor** アノテーションを付けなければなりません。 これらのアノテーションの両方かいずれかがインターセプタクラスに付けられていないと、 デプロイメントエラーが発生します。 また、 インターセプタは次のように **@Provider** アノテーションが付けられていなければなりません。

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor {

    public void write(MessageBodyWriterContext context) throws
IOException, WebApplicationException
    {
       context.getHeaders().add("My-Header", "custom");
       context.proceed();
    }
}
```

これは、 応答にヘッダ値を追加するサーバー側のインターセプタになります。 **@Provider** と **@ServerInterceptor** アノテーションが付けられています。 **MessageBodyReader** の実行後に応答がコミットされるため、 **context.proceed()** を呼び出す前にヘッダを変更する必要があります。



重要

context.proceed() を呼び出さないと呼び出しができません。

28.2. PREPROCESSINTERCEPTOR

org.jboss.resteasy.spi.interception.PreProcessInterceptor は、 JAX-RS リソースメソッドが見つかった後、メソッドが呼び出される前に実行されます。 サーバー上のみで使用できますが、 **@ServerInterceptor** アノテーションを付けなければなりません。 セキュリティ機能の実装や、 Java リクエストの先制に使用できます。 RESTEasy のセキュリティ実装は、 ユーザーが認証されないと、 これらのインターセプタを使用して呼び出しの前に要求を停止します。 RESTEasy のキャッシングフレームワークはこれらのインターセプタを使用してキャッシュされた応答を返し、 メソッドを複数回呼び出さないようにします。 インターセプタのインターフェースは次の通りです。

```
    public interface PreProcessInterceptor
    {
       ServerResponse preProcess(HttpRequest request, ResourceMethod
method) throws Failure, WebApplicationException;
    }
```

PreProcessInterceptor は順番に実行され、 実際の JAX-RS 呼び出しはラッピングしません。 図解すると次のようになります。

```
    for (PreProcessInterceptor interceptor : preProcessInterceptors) {
       ServerResponse response = interceptor.preProcess(request, method);
       if (response != null) return response;
    }
    executeJaxrsMethod(...);
```

preProcess() メソッドが **ServerResponse** を返す場合、 基礎の JAX-RS メソッドは呼び出されず、 ランタイムがその応答を処理してクライアントに返します。

28.3. POSTPROCESSINTERCEPTORS

org.jboss.resteasy.spi.interception.PostProcessInterceptor は **MessageBodyWriter** が呼び出された後、 JAX-RS メソッドが呼び出される前に実行されます。 サーバー側のみで使用でき、 **PreProcessInterceptor** との対称性を提供するため存在します。 **MessageBodyWriter** が呼び出されない可能性がある時に、 応答ヘッダを設定するために使用されます。 オブジェクトはラッピングせず、 **PreProcessInterceptor** と同様に順番通り呼び出されます。

```
    public interface PostProcessInterceptor
    {
       void postProcess(ServerResponse response);
    }
```

28.4. CLIENTEXECUTIONINTERCEPTORS

org.jboss.resteasy.spi.interception.ClientExecutionInterceptor クラスはクライアント側のみです。 **MessageBodyWriter** の後、 **ClientRequest** がクライアント側で構築された後に実行されます。 サーバーに送信された HTTP 呼び出しをラッピングします。 RESTEasy の GZIP サポートでは、 **Accept** ヘッダを設定し、 要求が送信される前に **gzip, deflate** が含まれるようにします。 RESTEasy のクライアントキャッシュでは、 リソースで試行を行う前にリソースがキャッシュに含まれているかを確認します。 これらのインターセプタには **@ClientInterceptor** と **@Provider** アノテーションが付けられなければなりません。

```
    public interface ClientExecutionInterceptor
    {
       ClientResponse execute(ClientExecutionContext ctx) throws
Exception;
    }

    public interface ClientExecutionContext
    {
       ClientRequest getRequest();

       ClientResponse proceed() throws Exception;
    }
```

MessageBodyReader と同様に、 **proceed()** を呼び出さないと呼び出しが停止されます。

28.5. バインディングインターセプタ

デフォルトでは、 登録されたインターセプタはいずれも要求ごとに呼び出されます。 これを変更するには、 インターセプタが **org.jboss.resteasy.spi.AcceptedByMethod** インターフェースを実装するようにします。

```
    public interface AcceptedByMethod
    {
```

```
        public boolean accept(Class declaring, Method method);
     }
```

インターセプタがこのインターフェースを実装すると、 RESTEasy は **accept()** メソッドを呼び出します。 このメソッドが **true** を返すと、 RESTEasy はこのインターセプタを JAX-RS メソッドの呼び出しチェーンに追加します。 **false** を返すと、 インターセプタは呼び出しチェーンへ追加されません。 例は次の通りです。

```
@Provider
@ServerInterceptor
public class MyHeaderDecorator implements MessageBodyWriterInterceptor,
AcceptedByMethod {

    public boolean accept(Class declaring, Method method) {
       return method.isAnnotationPresent(GET.class);
    }

   public void write(MessageBodyWriterContext context) throws IOException,
WebApplicationException
   {
      context.getHeaders().add("My-Header", "custom");
      context.proceed();
   }
}
```

この例では、 **@GET** アノテーションが JAX-RS メソッドに存在するか **accept()** メソッドが確認します。 存在する場合、 インターセプタはそのメソッドの呼び出しチェーンへ適用されます。

28.6. インターセプタの登録

インターセプタに **@Provider** アノテーションが付いていると、 **web.xml** の **resteasy.providers context-param** にリストされるか、 **Application.getClasses()** または **Application.getSingletons()** メソッド内のクラスやオブジェクトとして返されます。

28.7. インターセプタの順番と優先度

インターセプタによっては呼び出される順番の考慮が必要なものもあります。 例えば、 セキュリティインターセプタは常に最初に呼び出されなければなりません。 他のインターセプタの動作は、 ヘッダを追加するインターセプタが起因となることがあります。 デフォルトでは、 登録されたインターセプタの呼び出し順序をユーザーが制御することはできませんが、 インターセプタの **優先度** を指定することはできます。

インターセプタの優先度はインターセプタクラスを一覧表示することで指定されません。 **@org.jboss.resteasy.annotations.interception.Precedence** アノテーションによって特定のインターセプタクラスが **優先度ファミリ** に関連付けされます。 ファミリ構造で優先度を指定すると、 順番に影響する内蔵インターセプタを保護できるため、 設定を簡略化することができます。

以下は実行順にリストされたファミリです。

```
     SECURITY
     HEADER_DECORATOR
     ENCODER
```

```
REDIRECT
DECODER
```

優先度ファミリに関連付けされていないインターセプタは最後に呼び出されます。 通常、 **SECURITY** には **PreProcessInterceptor** が含まれます。 認証前の発生を最小限にするため、 これらのインターセプタを最初に呼び出す必要があります。 **HEADER_DECORATOR** は、 応答や発信要求にヘッダを追加するインターセプタです。追加されたヘッダが他のインターセプタの動作に影響することがあるため、 これらのインターセプタの優先度は 2 番目になります。 **ENCODER** インターセプタは **OutputStream** を変更します。 例えば、 GZIP インターセプタは **GZIPOutputStream** を作成し、 圧縮のため実際の **OutputStream** をラッピングします。 **REDIRECT** インターセプタは要求を再ルーティングし、 JAX-RS メソッドを迂回するため、 通常 **PreProcessInterceptor** で使用されます。 **DECODER** インターセプタは **InputStream** をラッピングします。 例えば、 GZIP インターセプタデコーダは **GzipInputStream** インスタンスの **InputStream** をラッピングします。

特定のファミリへカスタムインターセプタを関連付けるには、 **@org.jboss.resteasy.annotations.interception.Precendence annotation** アノテーションを付けます。

```
@Provider
@ServerInterceptor
@ClientInterceptor
@Precedence("ENCODER")
public class MyCompressionInterceptor implements
MessageBodyWriterInterceptor {...}
```

org.jboss.resteasy.annotations.interception パッケージには、 完全にタイプを安全化する便利なアノテーション **@DecoredPrecedence**、 **@EncoderPrecedence**、 **@HeaderDecoratorPrecedence**、 **@RedirectPrecedence**、 **@SecurityPrecedence** があります。 これらのアノテーションは **@Precedence** の代わりに使用します。

28.7.1. カスタム優先度

独自の優先度ファミリを定義し、 **@Precedence** アノテーションで適用することができます。

```
 @Provider
 @ServerInterceptor
 @Precedence("MY_CUSTOM_PRECEDENCE")
 public class MyCustomInterceptor implements MessageBodyWriterInterceptor
{...}
```

独自の便利なアノテーションを作成するには、 メタアノテーションとして **@Precedence** を使用します。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Precedence("MY_CUSTOM_PRECEDENCE")
public @interface MyCustomPrecedence {}
```

デプロイメント時に RESTEasy がエラーを表示するため、 カスタム優先度は登録するようにしてください。 次のように、 コンテキストパラメータを用いてカスタム優先度を登録することができます。

```
resteasy.append.interceptor.precedence
resteasy.interceptor.before.precedence
resteasy.interceptor.after.precedence
```

resteasy.append.interceptor.precedence は優先度ファミリをこのリストに追加します。 **resteasy.interceptor.before.precedence** は優先度の前となるファミリを指定できるようにします。 **resteasy.interceptor.after.precedence** は優先度の後となるファミリを指定できるようにします。 例は次の通りです。

```
<web-app>
    <display-name>Archetype RestEasy Web Application</display-name>

    <!-- testing configuration -->
    <context-param>
        <param-name>resteasy.append.interceptor.precedence</param-name>
        <param-value>END</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.interceptor.before.precedence</param-name>
        <param-value>ENCODER : BEFORE_ENCODER</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.interceptor.after.precedence</param-name>
        <param-value>ENCODER : AFTER_ENCODER</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/test</param-value>
    </context-param>

    <listener>
        <listener-
class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listene
r-class>
    </listener>

    <servlet>
        <servlet-name>Resteasy</servlet-name>
        <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</ser
vlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Resteasy</servlet-name>
        <url-pattern>/test/*</url-pattern>
    </servlet-mapping>

</web-app>
```

この **web.xml** ファイルでは 、**END** と **BEFORE_ENCODER**、 **AFTER_ENCODER** の 3 つの新しい優先度ファミリが定義されました。 この設定では、 ファミリの順番は次のようになります。

```
SECURITY
HEADER_DECORATOR
BEFORE_ENCODER
ENCODER
AFTER_ENCODER
REDIRECT
DECODER
END
```

第29章 非同期 HTTP 要求処理

非同期 HTTP 要求処理は、 個別のスレッド (希望する場合) で NIO (ノンブロッキング I/O: Non-blocking Input/Output) を使用して単一の HTTP 要求を処理できるようにします。 この機能は **COMET 機能** とも呼ばれます。 クライアントが遅延応答に対してサーバーをプルする場合が非同期 HTTP の主な使用例です。

クライアントとサーバーの両方からプッシュまたはプルする AJAX チャットクライアントが一般的な例です。 この場合、 新しいメッセージを待つ間、 クライアントがサーバーのソケットで長期間ブロックします。 同期 HTTPでは (サーバーが受信および発信 I/O でブロックする)、 クライアント接続毎に 1 つのスレッドが消費されるため、 メモリとスレッドの両方のリソースを消費します。 複数の同時クライアントがこのようにブロックする場合、 リソースは効率的に使用されずサーバーはうまくスケールしません。

Tomcat、 Jetty、 JBoss Web はすべて非同期 HTTP 要求処理に対して同様の (プロプライタリ) サポートを持っています。 この機能は現在 Servlet 3.0 仕様で標準化されています。 RESTEasy は、 簡単なコールバック API を提供し、 非同期の機能を備えています。 また、 Servlet 3.0 (Jetty 7より)、 Tomcat 6、 JBoss Web 2.1.1 との統合をサポートします。

RESTEasy の非同期 HTTP サポートは、 **@Suspend** アノテーションと **AsynchronousResponse** インターフェースの 2 つのクラスより実装されます。

```
public @interface Suspend
{
   long value() default -1;
}

import javax.ws.rs.core.Response;

public interface AsynchronousResponse
{
   void setResponse(Response response);
}
```

現在実行しているスレッドから HTTP 要求または応答を分離するべきで、 現スレッドは自動的に応答を処理するべきでないことを **@Suspend** は RESTEasy に伝えます。 **@Suspend** への引数は要求がキャンセルされるまでの時間 (ミリ秒単位) です。

AsynchronousResponse はコールバックオブジェクトで、 RESTEasy によってメソッドへ挿入されます。 アプリケーションコードは処理のため **AsynchronousResponse** を異なるスレッドへ移動します。 **setResponse()** を呼び出すとクライアントへ応答が返され、 HTTP 要求を終了します。 非同期処理の例は次の通りです。

```
import org.jboss.resteasy.annotations.Suspend;
import org.jboss.resteasy.spi.AsynchronousResponse;

@Path("/")
public class SimpleResource
{

   @GET
   @Path("basic")
   @Produces("text/plain")
   public void getBasic(final @Suspend(10000) AsynchronousResponse
response) throws Exception
```

```
   {
      Thread t = new Thread()
      {
         @Override
         public void run()
         {
            try
            {
               Response jaxrs =
Response.ok("basic").type(MediaType.TEXT_PLAIN).build();
               response.setResponse(jaxrs);
            }
            catch (Exception e)
            {
               e.printStackTrace();
            }
         }
      };
      t.start();
   }
}
```

29.1. TOMCAT 6 と JBOSS 4.2.3 のサポート

RESTEasy の非同期 HTTP API を Tomcat 6 や JBoss 4.2.3 と使用するには、 特別な RESTEasy Servlet を使用し、Tomcat (または JBoss 4.2.3 の JBoss Web) を設定して、NIO トランスポートを使用する必要があります。 最初に Tomcat (または JBoss Web) の **server.xml** ファイルを編集します。 **vanilla HTTP adapter** をコメントアウトし、 次を追加します。

```
<Connector port="8080" address="${jboss.bind.address}"
   emptySessionPath="true"
protocol="org.apache.coyote.http11.Http11NioProtocol"
   enableLookups="false" redirectPort="6443" acceptorThreadCount="2"
pollerThreadCount="10"
/>
```

デプロイされた RESTEasy アプリケーションも異なる RESTEasy Servlet **org.jboss.resteasy.plugins.server.servlet.Tomcat6CometDispatcherServlet** を使用する必要があります。 このクラスは、 **web.xml** の **async-http-tomcat-xxx.jar** (または **async-http-tomcat6** アーティファクト ID の Maven レポジトリ内) にあります。

```
<servlet>
   <servlet-name>Resteasy</servlet-name>
   <servlet-
class>org.jboss.resteasy.plugins.server.servlet.Tomcat6CometDispatcherServ
let</servlet-class>
</servlet>
```

29.2. SERVLET 3.0 のサポート

2008年10月20日現在、 Jetty 7.0.pre3 (mortbay.org) のみが未完成である Servlet 3.0 仕様の現草案をサポートしています。

デプロイされた RESTEasy アプリケーションも異なる RESTEasy Servlet **org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher** を使用する必要があります。 このクラスは、 **web.xml** の **async-http-servlet-3.0-xxx.jar** (または **async-http-servlet-3.0** アーティファクト ID の Maven レポジトリ内) にあります。

```
<servlet>
   <servlet-name>Resteasy</servlet-name>
   <servlet-
class>org.jboss.resteasy.plugins.server.servlet.HttpServlet30Dispatcher</s
ervlet-class>
</servlet>
```

29.3. JBOSSWEB と JBOSS AS 5.0.X のサポート

JBossWeb コンテナは JBoss AS 5.0.x に同梱され、 以降のバージョンには非同期 HTTP 処理を有効にする JBoss Native プラグインが必要となります。 詳細は JBoss Web のドキュメントを参照してください。

デプロイされた RESTEasy アプリケーションも異なる RESTEasy Servlet **org.jboss.resteasy.plugins.server.servlet.JBossWebDispatcherServlet** を使用する必要があります。 このクラスは、 **web.xml** の **async-http-jbossweb-xxx.jar** (または **async-http-jbossweb** アーティファクト ID の Maven レポジトリ内) にあります。

```
<servlet>
   <servlet-name>Resteasy</servlet-name>
   <servlet-
class>org.jboss.resteasy.plugins.server.servlet.JBossWebDispatcherServlet<
/servlet-class>
</servlet>
```

第30章 非同期ジョブサービス

RESTEasy の非同期ジョブサービスは、 O'Reilly の 『Restful Web Services』 で定義されている非同期ジョブパターンの実装です。 非同期性を同期プロトコルに追加するためのものです。

30.1. 非同期ジョブの使用

HTTP は非同期のプロトコルで、 非同期の呼び出しに対応できます。 HTTP 1.1 の応答コード **202** (許可) は、 サーバーが処理の応答を受信して許可されたものの、その処理は完了していないことを意味します。 RESTEasy の非同期ジョブサービスはこのタイプの応答を基にしています。

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

例えば、 **asynch** クエリパラメータを **true** に設定して上記を投稿した場合、 RESTEasy は **202** (許可) 応答コードを返し、 バックグラウンドで呼び出しを実行します。 また、 URL がバックグラウンドメソッドの応答の場所を示した状態で Location ヘッダを返します。

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

URI には次のようなフォームがあります。

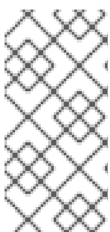
```
/asynch/jobs/{job-id}?wait={millisconds}|nowait=true
```

このジョブ URL で **GET**、**POST**、 **DELETE** 操作を実行できます。 **GET** はジョブが完了すると JAX-RS ソースメソッドの応答を返します。 ジョブが完了しなかった場合、 **GET** は応答コード **202** (Accepted) を返します。 **GET** の呼び出しはジョブを削除しないため、 複数回呼び出されることがあります。 RESTEasy のジョブキューが満杯になると、 最も長い時間使用されていないジョブをメモリより追放 (Evict) します。 手作業でキューを空にするには、 URI 上で **DELETE** を呼び出します。 **POST** は **JOB** 応答を読み取り、 完了後に **JOB** を削除します。

GET と **POST** はミリ秒単位の最大待機時間を指定できるようにします (**wait** クエリパラメータ)。 例は次の通りです。

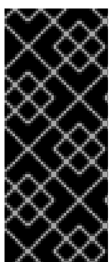
```
POST http://example.com/asynch/jobs/122?wait=3000
```

wait パラメータを指定しないと、 ジョブが完了しなかった場合に **GET** や **POST** は待機しません。



注記

GET、**DELETE**、 **PUT** メソッドを非同期で呼び出すことはできますが、 これらメソッドの HTTP 1.1 コントラクトを破ることになります。 これらの呼び出しが複数回呼び出された場合、リソースのステートは変更しないはずですが、 サーバーのステートを変更します。 POST メソッドを非同期で呼び出すようにしてください。



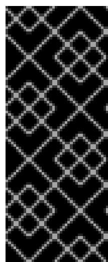
重要

RESTEasy のロールベースセキュリティ (アノテーション) は非同期ジョブサービスでは動作しません。 **web.xml ファイル内で XML 宣言型セキュリティを使用しなければなりません** 。 現在、ロールベースセキュリティを移植可能に実装することはできません。 将来的には固有の JBoss 統合が導入されるかもしれませんが、 他の環境はサポートしません。

30.2. 一方的: ファイアアンドフォーゲット

RESTEasy は **ファイアアンドフォーゲット** の概念もサポートします。 これは、 **202** (許可) を応答として返しますが、 ジョブは作成されません。 次のように **asynch** の代わりに **oneway** クエリパラメータを使用します。

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



重要

RESTEasy のロールベースセキュリティ (アノテーション) は非同期ジョブサービスでは動作しません。 **web.xml ファイル内で XML 宣言型セキュリティを使用しなければなりません** 。 現在、ロールベースセキュリティを移植可能に実装することはできません。 将来的には固有の JBoss 統合が導入されるかもしれませんが、 他の環境はサポートしません。

30.3. 設定と構成

非同期ジョブサービスはデフォルトでは有効になっていないため、 **web.xml** で有効にする必要があります。

```
<web-app>
    <!-- enable the Asynchronous Job Service -->
    <context-param>
        <param-name>resteasy.async.job.service.enabled</param-name>
        <param-value>true</param-value>
    </context-param>

    <!-- The next context parameters are all optional.
         Their default values are shown as example param-values -->

    <!-- How many jobs results can be held in memory at once? -->
    <context-param>
        <param-name>resteasy.async.job.service.max.job.results</param-
name>
        <param-value>100</param-value>
    </context-param>

    <!-- Maximum wait time on a job when a client is querying for it -->
    <context-param>
        <param-name>resteasy.async.job.service.max.wait</param-name>
        <param-value>300000</param-value>
    </context-param>

    <!-- Thread pool size of background threads that run the job -->
    <context-param>
        <param-name>resteasy.async.job.service.thread.pool.size</param-
name>
        <param-value>100</param-value>
    </context-param>

    <!-- Set the base path for the Job uris -->
    <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>
```

第31章 組み込みコンテナ

RESTEasy の JAX-RS には、 クラスパスから実行できる組み込み可能なサーバーが含まれています。このサーバーは TJWS (Tiny Java Web Server) の組み込み可能な Servlet コンテナを JAX-RS とパッケージ化します。

ディストリビューションより、 クラスパスに **resteasy-jaxrs.war/WEB-INF/lib** の **JAR** を移動します。 組み込みサーバーの **Registry** を使用して、 JAX-RS Bean をプログラムで登録しなければなりません。

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

   @GET
   public String get() { return "hello world"; }


   public static void main(String[] args) throws Exception
   {
      TJWSEmbeddedJaxrsServer tjws = new TJWSEmbeddedJaxrsServer();
      tjws.setPort(8081);
      tjws.getRegistry().addPerRequestResource(MyResource.class);
      tjws.start();
   }
}
```

サーバーは、ホスト非暗号化または SSL ベースのリソースのどちらかをホストできますが、両方はできません。 詳細は、 **TJWSEmbeddedJaxrsServer** とそのサブクラス **TJWSServletServer** に関する Java ドキュメントを参照してください。 TJWS の Web サイトにも多くの情報があります。

Spring を使用するには、 **SpringBeanProcessor** が必要です。 疑似コードの例は次の通りです。

```
   public static void main(String[] args) throws Exception
   {
      final TJWSEmbeddedJaxrsServer tjws = new TJWSEmbeddedJaxrsServer();
      tjws.setPort(8081);

      org.resteasy.plugins.server.servlet.SpringBeanProcessor processor =
new SpringBeanProcessor(tjws.getRegistry(), tjws.getFactory();
      ConfigurableBeanFactory factory = new XmlBeanFactory(...);
      factory.addBeanPostProcessor(processor);

      tjws.start();
   }
```

第32章 サーバー側疑似フレームワーク

RESTEasy は疑似フレームワークを備えているため、 組み込み可能なコンテナを使用しなくてもリソース上で直接呼び出すことができます。

```
import org.resteasy.mock.*;
...

      Dispatcher dispatcher = MockDispatcherFactory.createDispatcher();

      POJOResourceFactory noDefaults = new
POJOResourceFactory(LocatingResource.class);
      dispatcher.getRegistry().addResourceFactory(noDefaults);

      {
         MockHttpRequest request = MockHttpRequest.get("/locating/basic");
         MockHttpResponse response = new MockHttpResponse();

         dispatcher.invoke(request, response);


         Assert.assertEquals(HttpServletResponse.SC_OK,
response.getStatus());
         Assert.assertEquals("basic", response.getContentAsString());
      }
```

MockHttpRequest と **MockHttpResponse** に関連するメソッドの全一覧は、 RESTeasy Java ドキュメントを参照してください。

第33章 JAX-RS と RESTEASY のセキュリティ

RESTEasy は Servlet としてデプロイされるため、 標準的な **web.xml** の制約を使用して認証と承認を有効にしなければなりません。

web.xmlの URL パターンマッチングは制限されているため、 **web.xml** の制約と JAX-RS の互換性は制限されます。 **web.xml** の URL パターンは簡単なワイルドカードのみをサポートします。 例として、 次のような JAX-RS リソースがあるとしましょう。

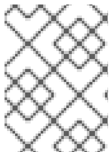
```
/{pathparam1}/foo/bar/{pathparam2}
```

この場合、 次のような **web.xml** URL パターンとしてマップできません。

```
/*/foo/bar/*
```

この問題を回避するには、 JAX-RS メソッド上で次のセキュリティアノテーションを使用します。 また、 **web.xml** に一般的なセキュリティ制約のエレメントを設定し、 認証を有効にしなければなりません。

RESTEasy JAX-RS は、 JAX-RS メソッド上で **@RolesAllowed**、 **@PermitAll**、 **@DenyAll** アノテーションをサポートします。 デフォルトでは、 RESTEasy はこれらのアノテーションを認識しません。 次のようにコンテキストパラメータを設定して、 RESTEasy でのロールベースセキュリティを有効にしなければなりません。



注記

EJB を使用している場合はこれを有効にしないでください。 RESTEasy の代わりに EJB コンテナがこの機能を提供します。

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

この方法では、 使用されるすべてのロールを RESTEasy JAX-RS **WAR** ファイルと JAX-RS クラスの両方で宣言し、 JAX-RS ランタイムによって処理される各 URL にロールがアクセスできるようにセキュリティ制約を確立しなければなりません。RESTEasy が正しく認証を行うと仮定します。

RESTEasy の認証は、メソッドが **@RolesAllowed** アノテーションが付けられているかチェックし、 **HttpServletRequest.isUserInRole** を実行します。 **@RolesAllowed** の 1 つがチェックに合格すると、 要求が許可されます。 合格しないと、 **401** (非認証) 応答コードと共に応答が返されます。

変更された RESTEasy WAR ファイルの例は次の通りです。 宣言された各ロールは、 RESTEasy Servlet によって制御される各 URL へのアクセスを許可されます。

```

<web-app>

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

```
      <context-value>true</context-value>
   </context-param>

   <listener>
      <listener-
class>org.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
   </listener>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-
class>org.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

   <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>Test</realm-name>
   </login-config>

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


</web-app>
```

第34章 EJB 統合

EJB (Enterprise Java Beans) と統合するには、 最初に EJB の公開されたインターフェースを編集する必要があります。 現在、 RESTEasy は単純で移植可能な EJB との統合のみ実現できます。 そのため、 RESTEasy の **WAR** を手作業で設定する必要があります。

EJB を JAX-RS リソースにするには、 次のようにステートレスセッション Bean の **@Remote** または **@Local** インターフェースにアノテーションを付けます。

```
@Local
@Path("/Library")
public interface Library {

   @GET
   @Path("/books/{isbn}")
   public String getBook(@PathParam("isbn") String isbn);
}

@Stateless
public class LibraryBean implements Library {

...

}
```

次に、 RESTEasy の **web.xml** にて **resteasy.jndi.resources <context-param>** を使用して手作業で EJB を RESTEasy に登録します。

```
<web-app>
   <display-name>Archetype Created Web Application</display-name>
   <context-param>
      <param-name>resteasy.jndi.resources</param-name>
      <param-value>LibraryBean/local</param-value>
   </context-param>

   <listener>
      <listener-
class>org.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
   </listener>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-
class>org.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>

</web-app>
```

現在、 EJB の移植可能な統合メソッドはこの方法のみです。 RESTEasy の今後のバージョンは JBoss AS と綿密に総合される予定です。 そのため、 手作業による登録や **web.xml** の編集が不必要になります。

RESTEasy で **EAR** と EJB を使用している場合、 次の構造が便利です。

```
my-ear.ear
|------myejb.jar
|------resteasy-jaxrs.war
       |
       ----WEB-INF/web.xml
       ----WEB-INF/lib (nothing)
|------lib/
       |
       ----All Resteasy jar files
```

すべてのライブラリを **WEB-INF/lib** より削除し、 共有の **EAR** ライブラリに配置するか、 RESTEasy **JAR** の依存関係をアプリケーションサーバーのシステムクラスパスに配置します (JBoss AS の場合、 **server/default/lib** に置きます)。

第35章 SPRING の統合

RESTEasy は Spring 2.5 と統合します (他の Spring 統合にも関心があるため、 ユーザーの情報提供を歓迎します)。

35.1. 基本的な統合

Maven ユーザーは resteasy-spring アーティファクトを使用する必要があります。それ以外のユーザーは、ダウンロードしたディストリビューションで jar が利用できます。

```
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-spring</artifactId>
    <version>whatever version you are using</version>
</dependency>
```

RESTEasy には独自の Spring **ContextLoaderListener** が含まれています。 これは、 **BeanFactory** によって Bean が作成された時に JAX-RS アノテーションを処理する RESTEasy 固有の **BeanPostProcessor** を登録します。 そのため、 RESTEasy は自動的に **@Provider** と Bean クラス上の JAX-RS リソースアノテーションにスキャンを行い、 JAX-RS リソースとして登録します。

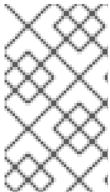
web.xml ファイルを変更する必要があります。

```
<web-app>
   <display-name>Archetype Created Web Application</display-name>

   <listener>
      <listener-
class>org.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-
class>
   </listener>

   <listener>
      <listener-
class>org.resteasy.plugins.spring.SpringContextLoaderListener</listener-
class>
   </listener>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-
class>org.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-
class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Resteasy</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>


</web-app>
```

SpringContextLoaderListener は **ResteasyBootstrap** によって初期化された ServletContext 属性を使用するため、 **SpringContextLoaderListener** は **ResteasyBootstrap** の後に宣言しなければなりません。

Bean ファクトリの作成に Spring **ContextLoaderListener** を使用しない場合、**org.jboss.resteasy.plugins.spring.SpringBeanProcessor** のインスタンスを割り振り RESTEasy **BeanFactoryPostProcessor** を手作業で登録することができます。**ResteasyProviderFactory** と **Registry** のインスタンスは、 **ServletContext** 属性や **org.resteasy.spi.ResteasyProviderFactory**、 **org.resteasy.spi.Registry** (これらクラスの完全修飾名ストリング) より取得できます。 また、 Servlet コンテキストから **Registry** と **ResteasyProviderFactory** へ自動的に参照を挿入する **org.jboss.resteasy.plugins.spring.SpringBeanProcessorServletAware** もあります。**RestasyBootstrap** を使用して RESTEasy をブートストラップしたと仮定します。

RESTEasy の Spring 統合は、 シングルトンと **プロトタイプ** スコープの両方をサポートし、**@Context** 参照の挿入に対応します。 しかし、 コンストラクタの挿入はサポートされません。 また、**プロトタイプ** スコープが使用されると、 RESTEasy は要求が送信される前に **@*Param** アノテーションが付けられたフィールドとセッタをすべて挿入します。



注記

RESTEasy の Spring 統合には自動的にプロキシされた Bean のみを使用できます。 ハンドコードされたプロキシ (**ProxyFactoryBean** による) を使用すると不具合が生じます。

35.2. SPRING MVC の統合

RESTEasy は Spring **DispatcherServlet** とも統合できます。これが機能的に有利な理由は、**web.xml** がより単純で、同じベース URL より Sping コントローラか RESTEasy のいずれかへ送信できるためです。さらに、 Spring **ModelAndView** オブジェクトを **@GET** リソースメソッドからの戻し引数として使用できます。 設定するには、 **web.xml** ファイルの Spring **DispatcherServlet** を使用し、 **springmvc-resteasy.xml** ファイルをベースの Spring Bean xml ファイルにインポートしなければなりません。 **web.xml** ファイルの例は次の通りです。

```
<web-app>
   <display-name>Archetype Created Web Application</display-name>

   <servlet>
      <servlet-name>Resteasy</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet;
</servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>Spring</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>


</web-app>
```

次に、 メインの Spring Bean xml ファイル内で **springmvc-resteasy.xml** ファイルをインポートします。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd
    http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
">

    <!-- Import basic SpringMVC Resteasy integration -->
    <import resource="classpath:springmvc-resteasy.xml"/>
....
```

第36章 GUICE 1.0 の統合

RESTEasy は Guice 1.0 との基本的な統合を実現できます。 RESTEasy は、 **@Path** と **@Provider** アノテーションに対する Guice モジュールのバインディングタイプをスキャンし、 これらのバインディングを登録します。 RESTEasy の **examples/** ディレクトリにある **guice-hello** プロジェクトにこの良い例があります。

```
@Path("hello")
public class HelloResource
{
   @GET
   @Path("{name}")
   public String hello(@PathParam("name") final String name) {
      return "Hello " + name;
   }
}
```

最初に、 JAX-RS リソースクラスを指定します。 この例では **HelloResource** です。 次にバインディングを定義する Guice モジュールクラスを作成します。

```
import com.google.inject.Module;
import com.google.inject.Binder;

public class HelloModule implements Module
{
    public void configure(final Binder binder)
    {
       binder.bind(HelloResource.class);
    }
}
```

これらのクラスを **WAR WEB-INF/classes** 内に置くか、 **WEB-INF/lib** 内の **JAR** に置きます。 次に **web.xml** ファイルを作成します。 次のように **GuiceResteasyBootstrapServletContextListener** を使用する必要があります。

```
<web-app>
    <display-name>Guice Hello</display-name>

    <context-param>
      <param-name>
        resteasy.guice.modules
      </param-name>
      <param-value>
        org.jboss.resteasy.examples.guice.hello.HelloModule
      </param-value>
    </context-param>

    <listener>
      <listener-class>

org.jboss.resteasy.plugins.guice.GuiceResteasyBootstrapServletContextListe
ner
      </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>
```

GuiceResteasyBootstrapServletContextListener は **ResteasyBootstrap** のサブクラスであるため、 **web.xml** ファイル内に別の RESTEasy 設定オプションを使用することができます。 **resteasy.guice.modules** コンテキストパラメータを見てください。 このパラメータには Guice モジュールであるクラス名のコンマ区切りリストを使用することができます。

第37章 クライアントフレームワーク

RESTEasy クライアントフレームワークは、 JAX-RS サーバー側仕様の代替となります。 JAX-RS アノテーションを使用して受信要求を RESTful Web サービスメソッドへマップする代わりに、 クライアントネットワークはリモート RESTful Web サービス上で呼び出す HTTP 要求を作成します。 リモート RESTful Web サービスは HTTP 要求を許可する任意のウェブリソースです。

RESTEasy には、 JAX-RS アノテーションを使用してリモート HTTP リソース上で呼び出しできるクライアントプロキシフレームワークがあります。 Java インターフェースを書いて、 メソッドやそのインターフェースに JAX-RS アノテーションを使用することができます。 例は次の通りです。

```
public interface SimpleClient
{
   @GET
   @Path("basic")
   @Produces("text/plain")
   String getBasic();

   @PUT
   @Path("basic")
   @Consumes("text/plain")
   void putBasic(String body);

   @GET
   @Path("queryParam")
   @Produces("text/plain")
   String getQueryParam(@QueryParam("param")String param);

   @GET
   @Path("matrixParam")
   @Produces("text/plain")
   String getMatrixParam(@MatrixParam("param")String param);

   @GET
   @Path("uriParam/{param}")
   @Produces("text/plain")
   int getUriParam(@PathParam("param")int param);
}
```

RESTEasy API は簡単で、 Apache HttpClient をベースにしています。 プロキシを生成し、 プロキシ上でメソッドを呼び出します。 呼び出されたメソッドは (メソッドのアノテーションを基に) HTTP 要求に変換され、 サーバーに掲載されます。 次のように設定を行います。

```
            import org.resteasy.plugins.client.httpclient.ProxyFactory;
            ...
            // this initialization only needs to be done once per VM

RegisterBuiltin.register(ResteasyProviderFactory.getInstance());


            SimpleClient client = ProxyFactory.create(SimpleClient.class,
"http://localhost:8081");
            client.putBasic("hello world");
```

その他のオプションは **ProxyFactory** Java ドキュメントを参照してください。 例えば、**HttpClient** 設定を微調整したいとしましょう。

@CookieParam はサーバーへ送信するクッキーヘッダを作成します。 独自の **javax.ws.rs.core.Cookie** オブジェクトを割り振り、 パラメータとしてクライアントプロキシメソッドへ渡すと、 **@CookieParam** は必要ありません。 サーバーへクッキーが渡されたことをクライアントフレームワークが認識するため、 追加のメタデータが必要ないからです。

クライアントフレームワークはサーバー上にある同じプロバイダを使用することができます。**addMessageBodyReader()** メソッドと **addMessageBodyWriter()** メソッドを使用して **ResteasyProviderFactory** シングルトンより手作業で登録する必要があります。

```
ResteasyProviderFactory.getInstance().addMessageBodyReader(MyReader.class)
;
```

37.1. 抽象応答

クライアント要求の応答コードか応答ヘッダにアクセスする必要がある場合、 **Client-Proxy** フレームワークは 2 つのオプションを提供します。

次のように、 メソッドコールより **javax.ws.rs.core.Response.Status** 列挙を返すことができます。

```
@Path("/")
public interface MyProxy {
   @POST
   Response.Status updateSite(MyPojo pojo);
}
```

サーバー上で呼び出した後、 クライアントプロキシは内部的に HTTP 応答コードを **Response.Status** 列挙へ変換します。

org.resteasy.spi.ClientResponse インターフェースを用いて要求に関連するすべてのデータを取得することができます。

```
/**
 * Response extension for the RESTEasy client framework. Use this, or
Response
 * in your client proxy interface method return type declarations if you
want
 * access to the response entity as well as status and header information.
 *
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public abstract class ClientResponse<T> extends Response
{
   /**
    * This method returns the same exact map as Response.getMetadata()
except as a map of strings
    * rather than objects.
    *
    * @return
```

```
    */
   public abstract MultivaluedMap<String, String> getHeaders();

   public abstract Response.Status getResponseStatus();

   /**
    * Unmarshal the target entity from the response OutputStream.  You
must have type information
    * set via <T> otherwise, this will not work.
    * <p/>
    * This method actually does the reading on the OutputStream.  It will
only do the read once.
    * Afterwards, it will cache the result and return the cached result.
    *
    * @return
    */
   public abstract T getEntity();

   /**
    * Extract the response body with the provided type information
    * <p/>
    * This method actually does the reading on the OutputStream.  It will
only do the read once.
    * Afterwards, it will cache the result and return the cached result.
    *
    * @param type
    * @param genericType
    * @param <T2>
    * @return
    */
   public abstract <T2> T2 getEntity(Class<T2> type, Type genericType);

   /**
    * Extract the response body with the provided type information.
GenericType is a trick used to
    * pass in generic type information to the resteasy runtime.
    * <p/>
    * For example:
    * <pre>
    * List<String> list = response.getEntity(new GenericType<List<String>
() {});
    * <p/>
    * <p/>
    * This method actually does the reading on the OutputStream.  It will
only do the read once.  Afterwards, it will
    * cache the result and return the cached result.
    *
    * @param type
    * @param <T2>
    * @return
    */
   public abstract <T2> T2 getEntity(GenericType<T2> type);
}
```

すべての **getEntity()** メソッドは呼び出されるまで延期されます。 よって、 これらのメソッドの 1 つを呼び出すまで応答 **OutputStream** は読み取られません。 メソッド宣言内に **ClientResponse** を

テンプレートした場合のみパラメータのない **getEntity()** メソッドを使用することができます。 RESTEasy はこのジェネリックタイプの情報を使用して **OutputStream** がアンマーシャリングされるメディアタイプを決定します。 パラメータを受け入れる **getEntity()** メソッドでは、 応答がマーシャリングされるオブジェクトタイプを指定することができます。 これにより、 ランタイム時に希望のタイプを動的に抽出することができます。 例は次の通りです。

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

   @GET
   @Produces("application/xml")
   ClientResponse<LibraryPojo> getAllBooks();
}
```

クライアントプロキシフレームワークが HTTP 応答ボディをアンマーシャリングできるようにするため、 **ClientResponse** のジェネリック宣言に **LibraryPojo** が含まれるようにします。

37.2. クライアントとサーバー間でインターフェースを共有する

通常、 クライアントとサーバー間でインターフェースを共有することは可能です。 前述の例では、 JAX-RS サービスはアノテーションが付けられたインターフェースを実装し、 クライアント側で呼び出しを行うため同じインターフェースを再使用してクライアントプロキシを作成しなければなりませんでした。ただし、 JAX-RS メソッドが応答オブジェクトを返す時は制限されます。 raw 応答の戻しタイプ宣言ではクライアントにタイプ情報がないため問題が発生します。 この問題を回避する方法は 2 つあります。 1 つ目の方法は、 **@ClientResponseType** アノテーションを使用することです。

```
import org.jboss.resteasy.annotations.ClientResponseType;
import javax.ws.rs.core.Response;

@Path("/")
public interface MyInterface {

   @GET
   @ClientResponseType(String.class)
   @Produces("text/plain")
   public Response get();
}
```

しかし、この方法では一部の **MessageBodyReader** や **MessageBodyWriter** は要求の一致や対応にジェネリックタイプの情報が必要なため、動作しないことがあります。

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

   @GET
   @Produces("application/xml")
   public Response getMyListOfJAXBObjects();
}
```

この場合、 クライアントコードは返された応答オブジェクトを **ClientResponse** へキャストし、 タイプされた **getEntity()** メソッドのうち 1 つを使用することができます。

```
MyInterface proxy = ProxyFactory.create(MyInterface.class,
"http://localhost:8081");
```

```
ClientResponse response = (ClientResponse)proxy.getMyListOfJAXBObjects();
List<MyJaxbClass> list = response.getEntity(new
GenericType<List<MyJaxbClass>>());
```

37.3. クライアントエラー処理

クライアントフレームワークを使用し、 プロキシメソッドが **ClientResponse** 以外を返す場合、 デフォルトのクライアントエラー処理が実行されます。**399** よりも値が大きい応答コードは自動的に **org.jboss.resteasy.client.ClientResponseFailure** の例外を発生させます。

```
   @GET
   ClientResponse<String> get() // will throw an exception if you call
getEntity()

   @GET
   MyObject get(); // will throw a ClientResponseFailure on response code
> 399
```

37.4. 手動の CLIENTREQUEST API

RESTEasy には要求を呼び出す手動の API **org.jboss.resteasy.client.ClientRequest** があります。 このクラスの詳細は Java ドキュメントを参照してください。 簡単な例は次の通りです。

```


   ClientRequest request = new
ClientRequest("http://localhost:8080/some/path");
   request.header("custom-header", "value");

   // We're posting XML and a JAXB object
   request.body("application/xml", someJaxb);

   // we're expecting a String back
   ClientResponse<String> response = request.post(String.class);

   if (response.getStatus() == 200) // OK!
   {
      String str = response.getEntity();
   }
```

第38章 MAVEN と RESTEASY

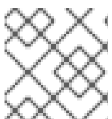
JBoss Maven レポジトリは http://repository.jboss.org/maven2 にあります。

次の **pom.xml** フラグメントと組み合わせることができます。 RESTEasy は様々なコンポーネントに分割されています。 必要に応じたコンポーネントを組み合わせることができます。 必ず *1.1.GA* を使用したい RESTEasy のバージョンに置き換えるようにしてください。

```
<repositories>
   <repository>
      <id>jboss</id>
      <url>http://repository.jboss.org/maven2</url>
   </repository>
</repositories>
<dependencies>
   <!-- core library -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jaxrs</artifactId>
      <version>1.1.GA</version>
   </dependency>

   <!-- optional modules -->

   <!-- JAXB support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-jaxb-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- multipart/form-data and multipart/mixed support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-multipart-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- Resteasy Server Cache -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-cache-core</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- Ruby YAML support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-yaml-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- JAXB + Atom support -->
   <dependency>
      <groupId>org.jboss.resteasy</groupId>
      <artifactId>resteasy-atom-provider</artifactId>
      <version>1.1.GA</version>
   </dependency>
   <!-- JAXB + Atom support -->
   <dependency>
```

```
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-atom-provider</artifactId>
         <version>1.1.GA</version>
      </dependency>
      <!-- Apache Abdera Integration -->
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>abdera-atom-provider</artifactId>
         <version>1.1.GA</version>
      </dependency>
      <!-- Spring integration -->
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-spring</artifactId>
         <version>1.1.GA</version>
      </dependency>
      <!-- Guice integration -->
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-guice</artifactId>
         <version>1.1.GA</version>
      </dependency>

      <!-- Asynchronous HTTP support with JBossWeb -->
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>async-http-jbossweb</artifactId>
         <version>1.1.GA</version>
      </dependency>

      <!-- Asynchronous HTTP support with Servlet 3.0 (Jetty 7 pre5) -->
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>async-http-servlet-3.0</artifactId>
         <version>1.1.GA</version>
      </dependency>

      <!-- Asynchronous HTTP support with Tomcat 6 -->
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>async-http-tomcat6</artifactId>
         <version>1.1.GA</version>
      </dependency>

   </dependencies>
```

また、**POM** をインポートして、 各モジュールバージョンの指定が必要がないようにすることもできます。



注記

これには Maven 2.0.9 が必要です。

```
    <dependencyManagement>
        <dependencies>
```

```
            <dependency>
                <groupId>org.jboss.resteasy</groupId>
                <artifactId>resteasy-maven-import</artifactId>
                <version<1.1.GA</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

第39章 JBOSS 5.X の統合

Resteasy 1.1.GA は JBoss Application Server との特別な統合がないため、 他のコンテナ同様に設定しインストールする必要がありますが、 若干の対処を必要とします。 WEB-INF/lib ディレクトリにある servlet-api-xxx.jar は問題発生の原因となることがあるため、 存在しないようにしてください。 また、JDK 6 を実行している場合、JAXB jar は JDK 6 に同梱されているためそれらをフィルタアウトするようにしてください。

第40章 旧バージョンからの移行

40.1. 1.0.X および 1.1-RC1 からの移行

最新バージョンの RESTEasy へ移行すると、 次のような変更が適用されます。

- 新しい **`resteasy.role.based.security context-param`** を使用して RESTEasy のロールベースセキュリティ (**`@RolesAllowed`**) を有効にできます。
- JAXB オブジェクトのリストやアレイ、 セットでは **`@Wrapped`** がデフォルトで有効です。 このアノテーションを使用して名前空間名やエレメント名を変更することもできます。
- **`@Wrapped`** は RESTEasy の名前空間プレフィックスで囲まれなくなり、 **`http://jboss.org/resteasy`** 名前空間ではなくデフォルトの名前空間が使用されるようになりました。
- **`@Wrapped JSON`** は簡単な JSON アレイで囲まれるようになりました。
- **`resteasy-jackson-provider-xxx.jar`** をクラスパスに置くと Jackson JSON プロバイダがトリガされるため、 これまで Jettison JAXB/JSON プロバイダを使用していた場合はコードエラーの原因となることがあります。 この問題を修正するには、 **`WEB-INF/lib`** またはクラスパスより Jackson を削除するか、 JAXB クラス上で **`@NoJackson`** アノテーションを使用します。
- **`tjws`** アーティファクトと **`servlet-api`** アーティファクトが **`resteasy-jar`** の依存関係で **provided** としてスコープされるようになりました。 **Class not found** エラーが発生する場合、 **pom** ファイル内でこれらのアーティファクトを **`provided`** または **`test`** としてスコープする必要があることがあります。

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>tjws</groupId>
    <artifactId>webserver</artifactId>
    <scope>provided</scope>
</dependency>
```

付録A 改訂履歴

改訂 5.1.2-2.400	2013-10-31	Rüdiger Landmann

Rebuild with publican 4.0.0

改訂 5.1.2-2	2012-07-18	Anthony Towns

Rebuild for Publican 3.0

改訂 5.1.2-100	Thu 8 December 2011	Russell Dickenson

JBoss Enterprise Application Platform 5.1.2 GA の変更が含まれます。このガイドの内容の変更については、『 リリースノート 5.1.2』 を参照してください。