REST Services White Paper

VERSION 2016

jade

Contents

REST Services

This white paper contains information about the REST-based Web services that JADE provides.

A Web service usually uses HTTP to exchange data. Unlike a Web application, which is typically HTML over HTTP, a Web service is Extensible Markup Language (XML) over HTTP. A client sends a request in XML, and the server responds with an XML response. This XML can be Plain Old XML (POX), which is typically a non-standard XML of which only the client and server can make sense, or it is standard Simple Object Access Protocol (SOAP).

A Representational State Transfer (REST) API is a service. A REST API differs from SOAP-based Web services in the way it is intended to be used. By using REST, the API tends to be lightweight and embraces HTTP. For example, a REST API leverages HTTP methods to present the actions a user would like to perform and the application entities would become resources these HTTP methods can act on. Although SOAP is not used, messages (requests and responses) are either in XML or JavaScript Object Notation (JSON).

Why REST Web Services?

REST-based Web services are implemented using HTTP. They offer a light-weight alternative to the original SOAP and WSDL-based Web services.

REST works with resources that are identified with a Uniform Resource Identifier (URI). REST resources are named with nouns as part of the URI rather than verbs; for example, **/customers** rather than **/getCustomers**. One of the key characteristics of RESTful Web API is that the URI or the request message does not include a verb.

To use REST services, a client sends an HTTP request using the **GET**, **POST**, **PUT**, or **DELETE** verb.

The traditional HTTP error messages (for example, *200 - OK* and *404 - Not found*) can be used to indicate whether a request is successful. If a request is successful, information can be returned in Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

Session handling is not performed, so there is no timeout of connections. Additionally, information is not retained between requests from a client. If that is required, it must be provided by the application developer.

REST Services in JADE

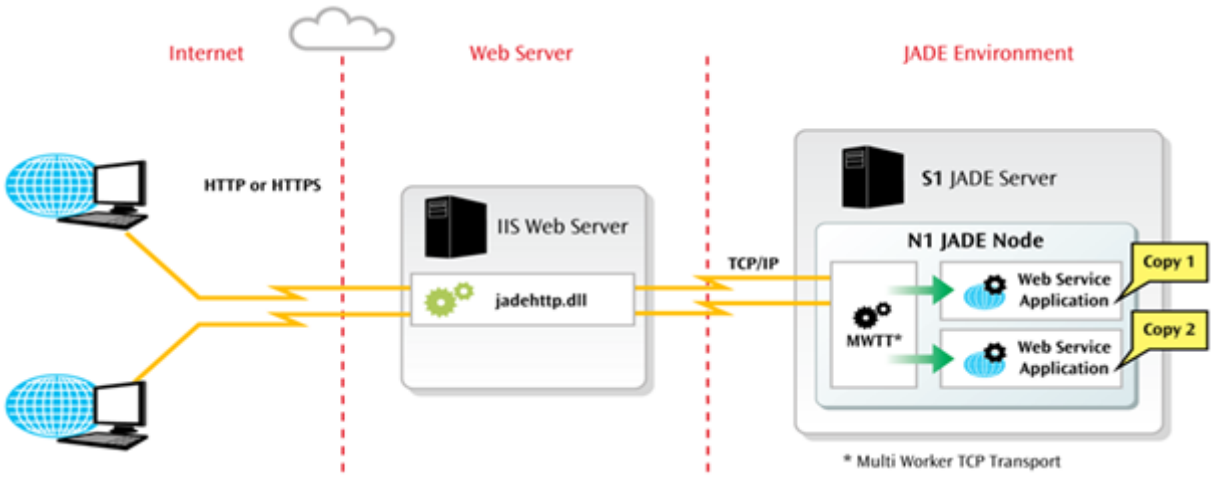
Earlier JADE releases supported only the Simple Object Access Protocol (SOAP) and WSDL-based Web services.

JADE now implements the Representational State Transfer (REST) stateless architecture style as a simpler alternative to SOAP Web services. Mainstream Web 2.0 service providers such as Google, Salesforce, and Facebook have endorsed this easier-to-use, resource-oriented model to expose their services. REST-based Web services, implemented using HTTP, offer a light-weight alternative to the Web services available in earlier releases.

JADE REST services currently support the following output formats.

- JSON (Microsoft JSON)
- XML (Microsoft XML (version .NET 4.5)
- JSONN (NewtonSoft JSON) version 6.0.1

REST services in JADE use the existing HTTP communications framework.



REST Service Components

This section contains:

The JadeRestService Class

A transient instance of a subclass of the **JadeRestService** class is created by each REST services application and is used by each REST services message that is received.

The **JadeRestService** class **processRequest** method is called on this object, passing the message details in the URL. That method decodes the URL and any objects passed in XML or JSON format, and calls the required method on the same **JadeRestService** object. The result returned by the method is encoded into XML or JSON, as requested. The **JadeRestService** class **reply** method is then called, passing the string to be returned to the client.

JadeRestService Class Methods

You can create REST service methods only for a **JadeRestService** subclass.

The methods defined in the **JadeRestService** class are summarized in the following table.

Method	Description
createVirtualDirectoryFile	Passes files created by a JADE application to the **jadehttp** library
deleteVirtualDirectoryFile	Deletes specified files from the virtual directory used by the **jadehttp** library
getOutputFormat	Returns an **Integer** value that represents the output format
getServerVariable	Returns the specified HTTP header information for your REST service request from the Internet Information Server (IIS)
isVDFilePresent	Returns **true** if the specified file is present in the virtual directory used by the **jadehttp** library
processRequest	Processes the received message
reply	Sends the returned value from the called method to the client

The following subsections contain examples of REST service methods to handle **GET**, **POST**, **PUT**, and **DELETE** requests, which could be defined in a **JadeRestService** subclass in your schema.

GET Method Example

The method in the following example returns a **Customer** object in XML or JSON in response to a **GET** request in which the customer identifier is specified.

```
getCustomer(pId: Integer): Customer updating;
vars
    customer: Customer;
begin
    // allCustomers is keyed on the customer id
    customer := app.myRoot.allCustomers.getAtKey(pId);
    if customer = null then
        // Setting HTTP status optional - you could simply return a 'null' customer
        self.httpStatusCode := 404;
        return null;
    else
        // Make an object to return and avoid returning references
        return customer.cloneSelf(true);
    endif;
end;
```

POST Method Example

The method in the following example creates a customer in response to a **POST** request in which the data for the customer is specified as primitive type parameters.

```
postCustomer(pName: String; pAddress: String);
vars
    customer: Customer;
begin
    beginTransaction;
    create customer;
    // Properties are set from the primitive parameters
    customer.name := pName;
    customer.address := pAddress;
```

```
    customer.myRoot := app.myRoot;
    commitTransaction;
end;
```

PUT Method Example

The method in the following example updates an existing customer in response to a **PUT** request.

Note One or more parameters are used to identify the **Customer** object to be updated. The remaining parameters are used to update the object.

```
putCustomer(pId: Integer; pName: String; pAddress: String);
vars
    customer: Customer;
begin
    // Identify customer to be updated using pId
    customer := app.myRoot.allCustomers.getAtKey(pId);
    if not customer = null then
        // Update customer using pName and pAddress
        beginTransaction;
        customer.name := pName;
        customer.address := pAddress;
        commitTransaction;
    endif;
end;
```

DELETE Method Example

The method in the following example deletes a specified customer in response to a **DELETE** request.

```
deleteCustomer(pId: Integer);
begin
    // Delete customer with specified id
    beginTransaction;
    delete app.myRoot.allCustomers.getAtKey(pId);
    commitTransaction;
end;
```

Syntax of a REST Request

A REST request is sent from a client as an HTTP verb (**GET**, **POST**, **PUT**, or **DELETE**), followed by the URL of the resource. The syntax is similar to that of other types of JADE Web-enabled applications.

```
Verb IIS server URL/jadehttp.dll/path[.xml|json|jsonn]?app_name[&extra_info]
     <-----------------------> <-------------------> <-------------------->
             first part              second part            third part
```

The following JADE REST service request retrieves information in JSON format for a customer with an identifier of **123**.

```
GET http://localhost/jade/jadehttp.dll/customer/123.json?RestApp
    <-------------------------------> <---------------> <----->
                first part                second part   third part
```

The first part of the URL is the path to the **jadehttp.dll** file.

```
http://localhost/jade/jadehttp.dll
```

In this example, the IIS host is the local machine and **jade** is an alias defined in IIS for the physical directory that contains the **jadehttp.dll** file.

The second part of the URL contains the following.

- Identifier of the resource, which in this example is **customer**.

  The **JadeRestService** method that is invoked for a **GET** request on the resource **/customer** is obtained by converting the HTTP verb to lowercase (**get**) and appending the name with the first letter capitalized (**Customer**), resulting in the method name **getCustomer**.

- Each additional URL path level is a parameter passed to the called method. Each string value is converted to the required method parameter type. An exception is raised if the data is invalid or there is a mismatch in the number of parameters.

  A **GET** request for **/customer/123** would result in a **getCustomer(123)** method call; that is, the **getCustomer** method would require the first parameter to be of the **Integer** type.

  A **GET** request for **/customer/Clark Kent** would result in a **getCustomer("Clark Kent")** method call; that is, the **getCustomer** method would require the first parameter to be of the **String** type.

  Note REST requests must be URL-encoded before the request is sent, so that **/customer/Clark Kent** would become **/customer/Clark%20Kent**.

  A **GET** request for **/customer/Clark Kent/Smallville** would result in a **getCustomer("Clark Kent", "Smallville")** method call; that is, the **getCustomer** method would require the type of the first and second parameters to be **String**.

  URL path levels separated by the slash character (**/**) are used to pass primitive parameters. An object parameter is passed as XML or JSON as the body of the data received. You can pass one object parameter only in a REST service request.

  A **ParamListType** parameter can be used in the method signature to receive multiple path parameters from the URL but it must be the last parameter of the JADE method. All parameters passed for a **ParamListType** parameter are assumed to be strings.

- You can include the output format of the data at the end of the path information.

  - **/customer/123.xml** returns customer information in Microsoft XML format
  - **/customer/123.json** returns customer information in Microsoft JSON format
  - **/customer/123.jsonn** returns customer information in NewtonSoft JSON format

  If the output format is not specified (**/customer/123**), data is returned in Microsoft JSON format.

The third part of the URL is the query string. It contains the name of the JADE REST services application. In the following example, the JADE REST services application is called **RestApp**.

To delete an employee:

```
DELETE http://localhost/jade/jadehttp.dll/customer/123?RestApp
```

To update the details of an employee:

```
PUT http://localhost/jade/jadehttp.dll/customer/123?RestApp
```

To create a new employee:

```
POST http://localhost/jade/jadehttp.dll/customer?RestApp
```

The XML or JSON user data is passed in as the body of the data received. That data is contained in the **httpIn** parameter passed to the **JadeRestService** class **processRequest** method.

To return a collection of all employees:

```
GET http://localhost/jade/jadehttp.dll/customers?RestApp
```

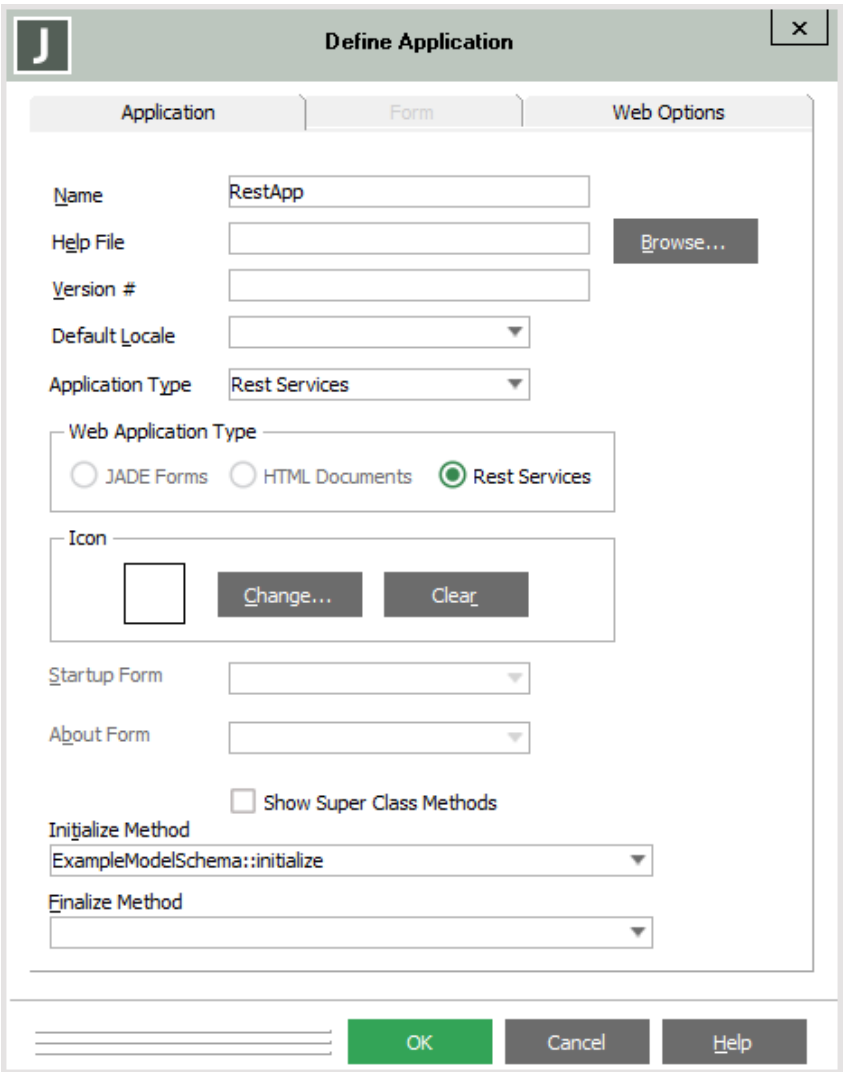
This request is passed using a **GET** HTTP request and returns an array of customers as XML or JSON.

See also "REST Message Handling", later in this document.

Defining a REST Service Application

The REST services application is defined in the Define Application dialog in the standard way. For details, see "Defining Applications, in Chapter 3 of the *JADE Development Environment User's Guide*.

On the **Application** sheet, select **Rest Services** or **Rest Services, Non-Gui** in the **Application Type** combo box.



This is defined from the development perspective. To successfully execute your application, set up the virtual directory and the **jadehttp.ini** (IIS) file correctly for your REST server. To configure IIS, see:

https://www.jadeworld.com/docs/jade-2016/Default.htm#resources/wp_erewhon/part_1/configuring_iis.htm

In the **jadehttp.ini** file, add an [*application-name*] section to enable clients to connect to the JADE REST services application. Set the parameter values to match the configuration information you specified on the Define Application dialog.

```
[RestApp]
ApplicationType=RestServices
TcpConnection=localhost
TcpPort=45000
```

REST Service Application Types

The **ApplicationType_Rest_Services** and **ApplicationType_Non_GUI_Rest** are available. These types are treated in most cases the same as **ApplicationType_Web_Enabled** and **ApplicationType_Non_GUI_Web**.

Use of the **ApplicationType_Rest_Services** type causes the display of the Web Application Monitor when the application is initiated; **ApplicationType_Non_GUI_Rest** does not.

JADE REST Services Application and SSL

You can implement security for REST-based Web services using:

- Operating system security and Internet Information Server (IIS) for data access
- Secure Sockets Layer (SSL) for data transmission

The communication from the client to IIS has no JADE involvement. Calling with an **https** header from C# automatically uses SSL. When the message is received by the JADE Rest Service, it has already been decrypted by IIS and passed to JADE as clear text.

To use SSL, you first need to establish the SSL configuration within IIS, which involves configuring the SSL certificate within IIS.

1. Add the certificate to the IIS Server certificates.
2. Create a binding on the default web site for HTTPS and the certificate.
3. Turn on SSL for the JADE web site that is to be used. If no certificate is required on the client, set the **Ignore** option for the client setting.

REST Message Handling

JADE handles REST messages as follows.

1. When the REST Services application is initiated, the web framework is initialized and the required number of application copies is activated.
2. A REST message is sent from a client using a URL that is of the form:

```
<iis-server-url>/jadehttp.dll/<path[.xml/json/jsonn]>?<application-name>
[&extra-info]
```

The following URL is an example of a REST message from the client.

```
http://localhost/jade/JadeHttp.dll/customer/123.json?RestApp
```

The path parameter contains:

- The name of the action at the first level; for example, customer. The method to be called is constructed from the type of request (get, put, post, or delete) and the action name with the first character capitalized; for example, **getCustomer**.
- Each additional URL path level becomes the parameters passed to the called method. Each string value is converted to the required method parameter type (an exception is generated if the data is invalid). For example, **123** is converted to an integer when the parameter is defined as an **Integer**.

  Note A **ParamListType** can be used in a method signature but it must be the last parameter.

- Optionally, the output format of the data returned can be included at the end of the path information; that is, .**xml**, .**json**, or .**jsonn**. If this is not present, the data is formatted into JSON. See step 7 later in this instruction for more details.

3. When a message arrives, the web processing framework calls the **JadeRestService** class **processRequest (httpIn: String; queryStr: String; pathIn: String; methodType: String);** method on the application REST service object. This method can be re-implemented by the application, if required, but that method must call **inheritMethod** for the processing to be completed.
4. When the **JadeRestService** class **processRequest** method is called:
   - It parses the <*path*> part of the received data. The first identifier is combined with the message method type (that is, **GET**, **PUT**, **POST**, or **DELETE** in lowercase) to create the method name to be called. For example, **/customer** for a **GET** type calls a **getCustomer** method (the first path character is made uppercase) on the **JadeRestService** subclass being used. An exception is generated if the method does not exist on the **JadeRestService** subclass.
   - Converts subsequent levels of the path into the parameters (validated and converted to the correct type) passed to the method. For example, **/customer/123/12:45** results in a call to **getCustomer(123, 12:45)** if the signature is **id: Integer; time: Time**.
   - If the method signature includes an object parameter:
     - The **httpIn** text is searched for an XML or JSON script. Such a script must be the last text part of the string.
     - An XML script is recognized by its **<?xml** header. The XML is parsed to create the contained object or objects. The XML can indicate that the passed object is null.

       If the base object is not null, the object must be of the type required by the parameter; otherwise an exception is generated.
     - If an XML header is not found, the text is searched for the first **{** if the parameter object type is not a collection or the first **{** or **[** when the parameter object type is a collection. If found, JSON format is assumed.

       JSON has no other header, and does not indicate the content type of object.

       The JSON format from NewtonSoft is also different from that of Microsoft.

       If the **{** or **[** character is not found and the end of the string is **''** (that is, null), the base object passed is assigned as being a null object.

If the JSON header is found, the JSON is parsed and the object defined by the parameter type is created and populated.

Note Because there is no class name in the JSON, passing the wrong object results in the entire content being ignored unless both object types happened to have the same property name.

- Any classes or properties referenced in the XML or JSON that do not exist in the schema are ignored.
- For properties that exist, the passed value in the XML or JSON is validated for its type and generates an exception if the format is invalid.
- Any text prior to the found XML or JSON script is ignored. This could be used by the application to pass additional information that could be processed by a reimplementation of the **JadeRestService** class **processRequest** method.
- One object parameter only can be defined for the method. It can appear in any position in the signature other than after a **ParamListType**.
- Any objects created by the XML or JSON parsing are deleted after the required method is called.
- The created base object (or null, if null was indicated) is passed as the object parameter.

5. An exception is generated if the path and any passed object do not match the method signature.
6. The located method is called, executing the logic defined by the application for the operation.
7. The method returns a value to be passed back to the client. This value is encoded into a string according to the format requested by the client.
   - The XML format is that expected by the Microsoft **DataContractSerializer** class for an **Object** or a **TimeStampInterval** primitive type return value. While other primitive type return values are formatted for the Microsoft **XmlSerializer** (**XmlSerializer** does not support the **TimeSpan** type). The output includes oids for each included object and references to already included objects, therefore supporting circular and multiple references to the same object in the returned data.
   - The JSON format is that expected by the Microsoft **DataContractJsonSerializer** class. This format type does not support circular references or multiple references to the same object in the returned data (an exception is generated if the situation is detected). This is the default format if no format was specified.
   - The JSONN format is that expected by the NewtonSoft JSON class software. This format is different from that of Microsoft in the structure, tags, and the format of some primitive types. The output includes oids for each object and references to already included objects, and therefore supports circular and multiple references to the same object in the returned data.

   Note also about return value handling:

   - If the returned value is an object, the entire referenced object tree is encoded into XML or JSON, as required.
   - For returned objects, all property values are included, including null values.
   - If the returned value is a string that is already encoded in XML format (that is, it starts with **<?xml**), the XML string value is sent as is.
8. The **JadeRestService** class **reply(str: String);** method is called, passing the string constructed from the returned value. This method sends the string back to the client. The **reply** method can be re-implemented by the application, but the re-implementation must call **inheritMethod**.

9. The **processRequest** method then deletes:
   - Any objects created from the passed XML or JSON.
   - The returned value, if it is a non-shared transient object.
   - Any objects added to the **objectsToBeDeleted** collection by user logic. (They must be non-shared transient objects; otherwise an exception is generated.)

     A returned shared transient object is not deleted on completion of the REST Services processing. It would be unsafe to do so, because JADE cannot be certain of whether that is the intention and JADE would have to go into transaction state to do so.

     Note Anything added to the **JadeRestService** class **objectsToBeDeleted** collection is expected be to non-shared transients, and any other object lifetime will cause the logic to fail because the logic is not in transient state.

JADE versus C# Types

The following table lists the C# type expected for each JADE type.

JADE Type	C# Type
Binary	Byte []
Boolean	Boolean
Byte	Byte
Character	Char
Date	DateTime
Decimal	Decimal
Integer	Int32
Integer64	Int64
JadeBytes	Byte []
HugeStringArray	String []
Point	String format <integer>, <integer>
Real	Double
String	String
StringUtf8	String
Time	DateTime
TimeStamp	DateTime
TimeStampInterval	TimeSpan
TimeStampOffset	DateTime (with UTC offset set)

Generating and Parsing JSON independent of REST

The **JadeJson** class enables JSON to be parsed or serialized as a standalone feature that is independent of the Representational State Transfer (REST) Application Programming Interface (API).

The methods defined in the **JadeJson** class that enable you to create, load, unload, and parse JSON in the same way you can with XML are summarized in the following table.

Method	Description
generateJson	Generates JSON from a primitive type variable or an object
generateJsonFile	Generates JSON from a primitive type variable or an object and writes the output to a file
parse	Parses JSON text to create and populate an object and all referenced objects
parseFile	Reads and parses JSON text from a file to create and populate an object and all referenced objects
parsePrimitive	Parses JSON text for a primitive type and returns the primitive type value
parsePrimitiveFile	Parses JSON text for a primitive type from a file and returns the primitive type value

Received and Reply Format

The received format of REST service messages is:

- XML

  XML format expected by the Microsoft **DataContractSerializer** for objects or **XmlSerializer** for primitive types (except for the **TimeStampInterval** primitive type). Objects are encoded with oids and circular references are allowed.

- JSON

  JSON format expected by the Microsoft **DataContractJsonSerializer**. No duplicate references are allowed.

- JSONN

  JSONN as expected by NewtonSoft. Objects are encoded with oids and circular references are allowed.

The reply format of REST service messages is:

- All object properties are returned, including null values.
- For objects, the entire object tree is returned.
- If the called method returns a string that is XML (that is, starts with **<? xml**), the string is sent as is.

The REST message limits and default values are:

- The **JadeDynamicObject** type is not supported by JADE REST services.
- REST service messages can contain only public and read-only properties that are going to be serialized. Protected properties are excluded from the serialization process.

Notes about REST Service Messages

When using the JADE REST service:

- A REST service is stateless; there is no session object.
- There is no time-out facility, because there is no session object.
- The standard web XML configuration file can be used.

- REST services do not provide an exposure facility. The application controls what object types can be passed from the client by the signatures of the methods called to handle the incoming requests. However, clicking the **Generate Description** button on the **Web Options** sheet of the Define Application dialog displays a common Save As dialog that enables you to specify or select the name and location of the .**xml** file to which the XML description of the REST service is written.

  The generated XML file is based on the following entities.

  - The application name and required URL.
  - The list of available resources with the:
    - HTTP name required (that is, **GET**, **PUT**, **POST**, or **DELETE**)
    - Resource id (for example, **Person**)
    - Required parameter names and types
    - Type of object required in JSON or XML format
  - The list of classes referred to by the signatures of the resource methods.
    - The name of the class
    - The name and C# type of each property of each class
- One object only can be passed to a method, but that method can contain references to other child objects that were also passed in the XML or JSON. The object parameter can occur in any location within the method signature except after a **ParamListType**.
- Any primitive type parameters for the called method must be specified in the URL path in the order in which they appear in the method signature.
- The web server in a REST Web service can be IIS or Apache.
- The **JadeRestService** class **processRequest** method, called to process the request, can be re-implemented.

  If the **processRequest** method is reimplemented, **inheritMethod** should always be called to complete the processing.

```
processRequest(httpIn: String; queryStr: String; pathIn: String;
               methodType: String);
```

- The **JadeRestService** class **reply** method, called by the framework to send the reply, can also be re-implemented.

```
reply(msg: String);
```

- Failures that are generated by IIS (for example, when the service is not available) are in HTML format.
- JADE logic and processing exceptions generate an XML reply of the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<Fault>
    <errorCode>11102</errorCode>
    <errorItem>Method 'TestRestService::deleteStringArray' not
               found</errorItem>
    <errorText>Requested Rest Service method does not exist</errorText>
</Fault>
```

  The XML style is required because JSON does not identify the name of the entity being returned.

- User logic can set the **JadeRestService** class **httpStatusCode** property to a value that will generate a WebException in the calling client C# logic (if not 0 and < 200 or > 299).

  The returned value from the called method is still also returned. The client C# logic can retrieve that information by processing the C# WebException **Response** property.

WADL-like Generated XML Description

The following is an example of a Web Application Description Language (WADL) generated XML description.

```
<Application name="RestTest"
    uri="http://localHost/Jade/jadehttp.dll/">
    <resources>
        <method name="GET" id="Person1">
            <request>
                <param name="id" type="int" />
            </request>
            <response type="Person" />
        </method>
        <method name="PUT" id="Person">
            <request>
                <param name="id" type="int" />
                <xml-or-json-object name="Person" />
            </request>
            <response type="Person" />
        </method>
        <method name="GET" id="PersonArray">
            <request/>
            <response type="List<Person>" />
         </method>
    </resources>
    <CommunicationClasses>
        <Person>
            <countryOfBirth type="Country" />
            <dob type="DateTime" />
            <forenames type="String" />
            <sex type="Char" />
            <surname type="String" />
        </Person>
        <PersonSub superclass="Person">
            <description type="String" />
        </PersonSub>
    </CommunicationClasses>
</Application>
```