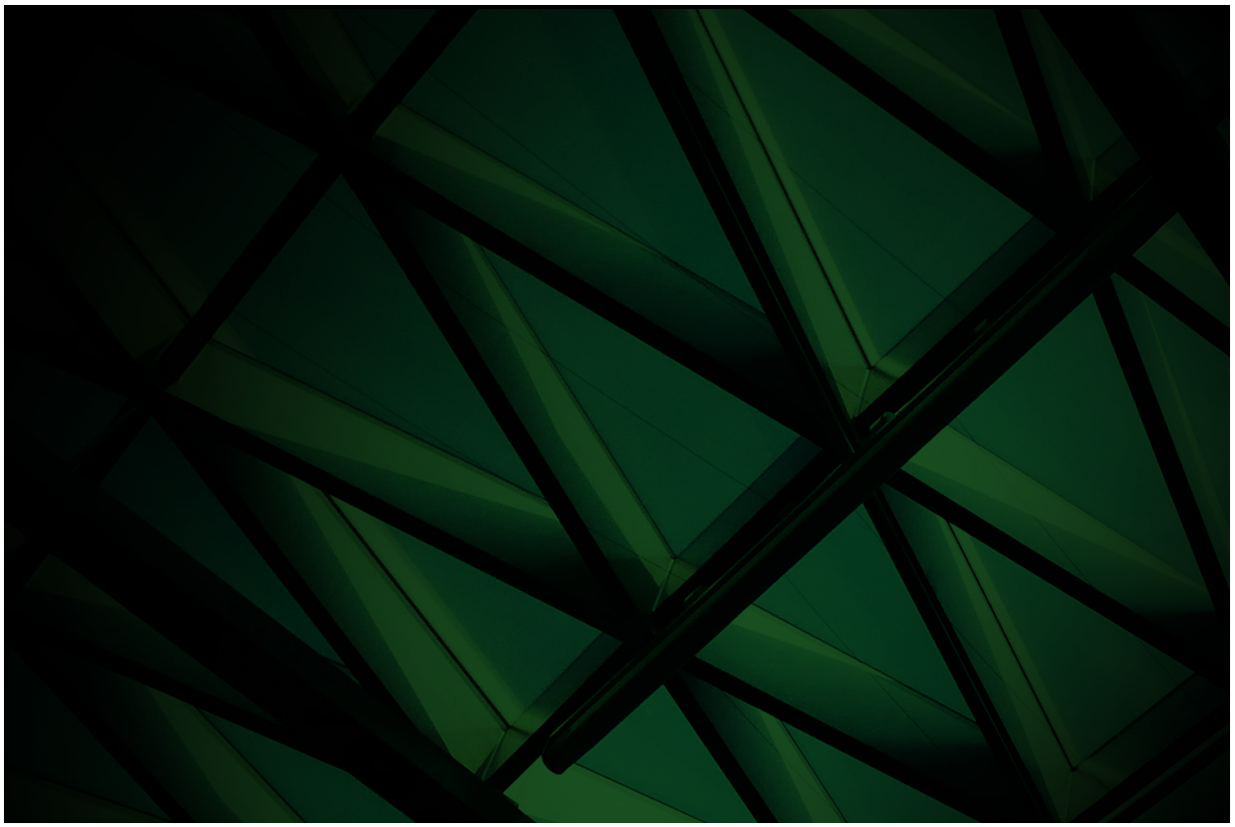


REST Services White Paper

VERSION 2022

jade platform

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 uses HTTP with a file format specialized for machine-readability; for example, Extensible Markup Language (XML) or JavaScript Object Notation (JSON). When a client sends a request in XML or JSON, the server responds with a response in the same format.

A Representational State Transfer (REST) API is a type of web service. A REST API differs from the older 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, with the application entities becoming resources on which these HTTP methods can act.

Why REST Web Services?

REST-based web services offer a light-weight alternative to the original SOAP and WSDL-based web services. As they are stateless, with the REST provider not storing any state for each client, REST-based web services are more scalable than previous web services implementations.

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.

As session handling is not performed, 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.

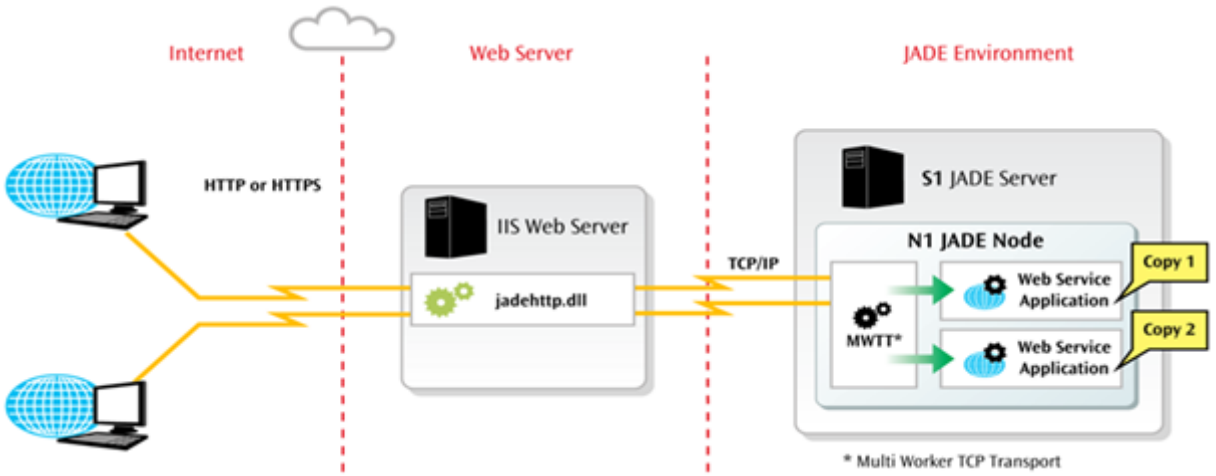
REST Services in Jade

Jade 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
```

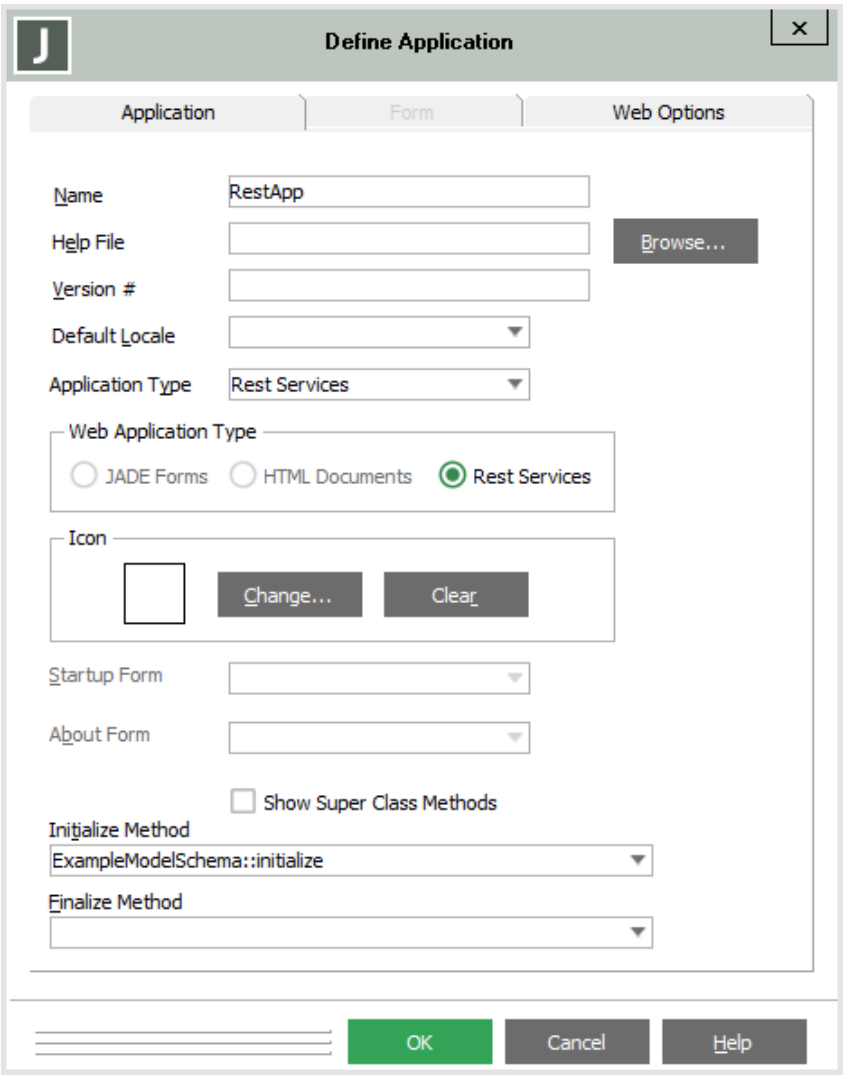
This request is passed using a **GET** HTTP request and returns an array of customers as XML or JSON.

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 *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://secure.jadeworld.com/JADETech/JADE2022/OnlineDocumentation/#resources/wperewhon/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 website for HTTPS and the certificate.
3. Turn on SSL for the Jade website that is to be used. If no certificate is required on the client, set the **Ignore** option for the client setting.

REST Message Handling

REST service messages can serialize public and protected properties. Protected properties can optionally be excluded from the serialization process, and read-only properties are always excluded from the serialization process.

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** 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.

   The **processRequest** method automatically receives the following parameter values from the web processing framework.

Parameter	Description
httpIn	Used for passing object parameters to method that require them. If a serialized object (in JSON or XML format) has been included in the HTTP request body, it is used. If not, the query string in the URL is used (that is, this will have the same value as the value of the **queryStr** parameter). If neither contains a serialized object, exception 11105 (*The object parameter required by called Rest Services method was not supplied*) is raised.
queryStr	Contains the query string; that is, the application name and optionally a single serialized object.
pathIn	Contains the resource name, any passed parameters, and the response format (if included); for example, **/Customer/123.json**.
methodType	The HTTP verb used to access the resource; that is, one of **GET**, **PUT**, **POST**, or **DELETE**.

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 object serialized in XML JSON, or JSONN format. This serialized object must be the last text part of the string.
  - An XML string 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, JSON, or JSONN that do not exist in the schema are ignored.
  - For properties that exist, the passed value in the XML, JSON, or JSONN is validated for its type and generates an exception if the format is invalid.
  - Any text prior to the found XML, JSON, or JSONN string 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, JSON, or JSONN 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, as follows.
   - For XML, the format depends on the returned type. If the return value is an **Object** class or a **TimeStampInterval** primitive type, the XML format is that which is expected by the Microsoft **DataContractSerializer** class.

Note The output can support circular and multiple references to the same object in the returned data, as it will include oids for all included and referenced objects. However, if the return value is any primitive type other than **TimeStampInterval**, the XML format is that which is expected by the Microsoft **XmlSerializer** class.

- For JSON, the format is that which is 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 (that is, an exception is generated if the situation is detected).

  Note This is the default format if no format was specified

- For JSONN (NewtonSoft JSON), the format is that which is expected by the NewtonSoft JSON class software. This format differs 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, JSON, or JSONN.
   - 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 transaction state.

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 **JadeJson** class contains the following methods, which enable you to create, load, unload, and parse JSON in the same way you can with XML.

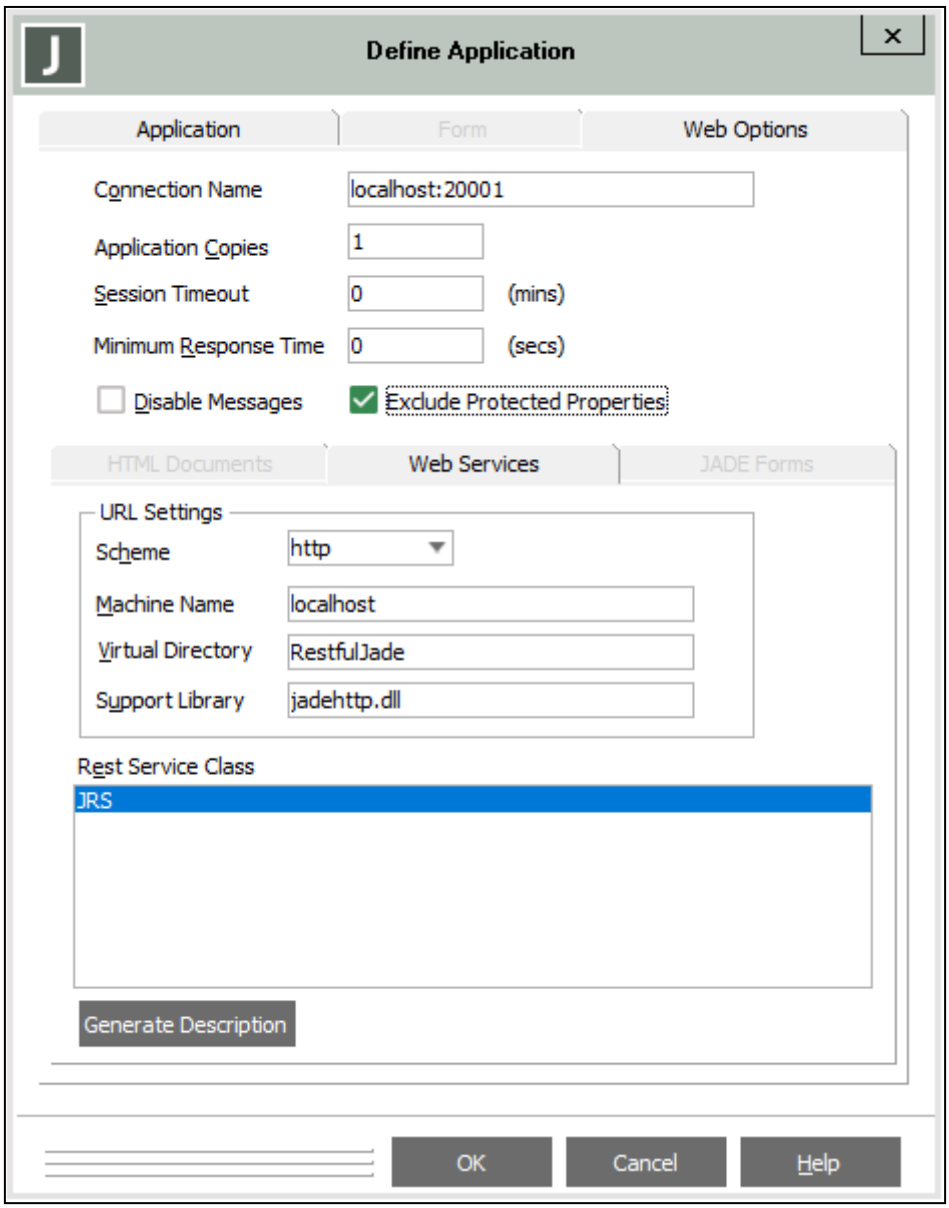
Method	Description
generateJson	Generates JSON from a primitive type variable or an object
generateJsonFile	Generates JSON from a primitive type variable or an object and writes the output to a file
parse	Parses JSON text to create and populate an object and all referenced objects
parseFile	Reads and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

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.
- The **JadeDynamicObject** type is not supported by Jade REST services.
- By default, all properties are serialized for objects returned by Jade REST service messages (whether public, protected, or read-only).
- By default, data received for read-only properties is ignored. Optionally, data received for protected properties is also ignored.

- To exclude protected properties for serialization and the receiving of data, check the **Exclude Protected Properties** check box on the **Web Options** sheet of the Define Application dialog.



- 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. For an example, see "WADL-like Generated XML Description", later in this document.

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.
- By default, 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>
```

If you want to change the exception output format at run time, specify the required **OutputFormat_** constant value in the **exceptionFormat** property of the **JadeRestService** class; for example, you could add **self.exceptionFormat := 0;** to the **create** method of your **JadeRestService** subclass. The class constant values for the exception output format are listed in the following table.

JadeRestService Class Constant	Value	Exceptions are in...
OutputFormat_Json	0	JSON (Microsoft JSON) format
OutputFormat_Xml	1	XML format (the default)
OutputFormat_Json_NewtonSoft	2	JSONN (NewtonSoft JSON) format

Any other value (for example, **-1**) means that exceptions are returned in the format controlled by the received URL.

- Jade logic and processing exceptions generate a response formatted in the style specified in the URL. When the output format is:
  - JSON (Microsoft JSON), the response is formatted as follows.

```
{"__type":"Fault",
"errorCode":"<error number>",
"errorItem":"<Exception type>",
"errorText":"<exception description>"}
```

  - JSONN (NewtonSoft JSON), the response is formatted as follows.

```
{"$type":"Fault",
"errorCode":"<error number>",
"errorItem":"<Exception type>",
"errorText":"<exception description>"}
```

  The following is an example of a generated XML reply.

```
<?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>
```

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

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

WADL-like Generated XML Description

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

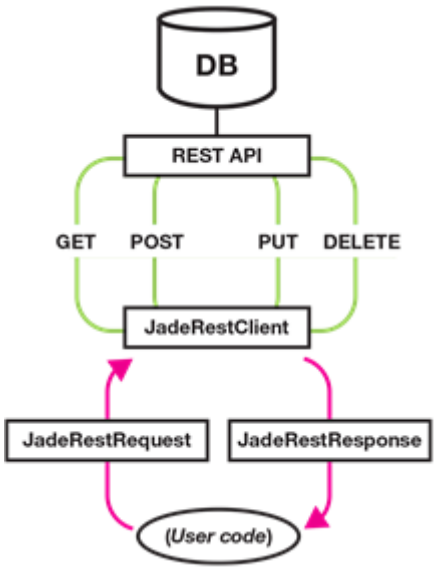
```
<Application name="RestTest"
    uri="http://localHost/Jade/jadehttp.dll/">
    <resources>
        <method name="GET" id="Person1">
            <request>
                <param name="id" type="int" />
            </request>
            <response type="Person" />
        </method>
        <method 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>
```

Jade REST Client

This section contains:

The JadeRestClient Class

The **JadeRestClient** class represents the REST client itself. It is this class that contains the logic for generating the appropriate Uniform Resource Identifiers (URIs) and headers for the REST request, and establishing the Hypertext Transfer Protocol (HTTP) connection to the server. It depends on two additional classes: the **JadeRestRequest** class, which represents a specific request that will be made to the server; and the **JadeRestResponse** class, which represents the response the server made to a specific request.



JadeRestClient Class Methods

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

Method	Description
create	Creates the **JadeRestClient** object
deleteResource	Executes a **DELETE** operation on a resource of the REST API specification
execute	Executes the specified operation on a resource of the REST API specification
get	Executes a **GET** operation on a resource of the REST API specification
post	Executes a **POST** operation on a resource of the REST API specification
put	Executes a **PUT** operation on a resource of the REST API specification
setEndpoint	Sets the endpoint, which represents the location of the REST API specification

Setting the Endpoint

A REST endpoint is the first part of the URI and it specifies the location at which the various resources of the API can be found.

When instantiating a **JadeRestClient** object, you need to provide the endpoint of the API as a parameter to the **create** method. For example, consider the following URI from the Swagger Petstore example REST API.

The **https://petstore.swagger.io/v2/** part is the endpoint, which is where the API is found, and it contains multiple resources. The **/pet/3** part is the specific resource we are accessing. Together they make up the URI; however it is the endpoint part we need to provide in the **create** method of the **JadeRestClient** object, as follows.

```
client := create JadeRestClient("https://petstore.swagger.io/v2/");
```

If you want to change the endpoint of an existing **JadeRestClient** object, you can use the **setEndpoint** method of the **JadeRestClient** class to do so, as follows.

```
client.setEndpoint("https://petstore.swagger.io/v2/");
```

The JadeRestRequest Class

The **JadeRestRequest** class represents a specific request that is to be made to the server.

You can use the methods of this class to customize the request before sending it through the **JadeRestClient** object to the server. For example, you can specify authorization headers, add object parameters in a variety of formats, and specify any required query strings.

JadeRestRequest Class Methods

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

Method	Description
addBearerToken	Adds a bearer token (for example, a JSON Web Token) to the REST request.
addFormUrlData	Adds object properties as (key, value) pairs when consuming a REST service that requires the **application/x-www-form-urlencoded** content-type.
addMultipartFormData	Adds content when consuming a REST service that requires the **multipart/form-data** content-type.
addObjectParam	Adds an object to the HTTP body when consuming a REST service that requires the **application/json** or **application/xml** content-type. The object is serialized into JSON or XML, depending on the content-type.
addQueryString	Adds a parameter to the query string part of the URI.
addURLSeg	Adds a value for one of the parameters contained within the REST API URI.
create	Instantiates an object of the **JadeRestRequest** class.
replaceDefaultDiscriminator	Replaces the default value of the discriminator with a new value.

Setting the Resource Location

A REST resource location is the second part of the URI and it specifies which resource of the API is to be accessed.

When instantiating a **JadeRestRequest** object, you need to provide a resource of the API as a parameter to the **create** method. For example, consider the following URI from the Swagger Petstore example REST API.

The **/pet/3** part is the specific resource we are accessing, which is the resource part we need to provide in the **create** method of the **JadeRestRequest** object, as follows.

```
client := create JadeRestRequest("/pet/3");
```

In this case, part of the resource location is a parameter to let the server know which of the **Pet** resource objects we are trying to access, specifically the **Pet** with an ID equal to **3**. We can make this more readable by using a parameter name in the **create** method and then using the **addURLSeg** method of the **JadeRestRequest** class, as follows.

```
client := create JadeRestRequest("/pet/{petID}");
client.addURLSeg("petID", "3");
```

Adding Authorization Headers

Some REST APIs will be secured and may require authorization to access some resources. The usual way of authorizing a request is by using a bearer token (for example, a JSON Web Token). You can add a token to the **JadeRestRequest** object by using the **addBearerToken** method, as follows.

```
request.addBearerToken(token);
```

If any token has been added with the **addBearerToken** method, an Authorization: Bearer HTTP header will be added to the request and the most-recent token will be used. (You never need to include more than one authorization bearer token.)

Adding Object Parameters

When making a **PUT** or **POST** request, you will need to provide a serialized object as part of the request; that is, the object to be PUT at the existing resource or POSTed to the new resource.

The format in which to serialize the object will vary, depending on the requirements of the API. As such, the **JadeRestRequest** class provides a variety of serialization formats. For details, see the following subsections.

JSON or XML Serialization

The most-common way to provide an object for a **PUT** or **POST** request is to include it in the body of the HTTP request, usually serialized into JSON or sometimes into XML format. You can use the **JadeRestRequest** class **addObjectParam** method to serialize an object into JSON or XML format, depending on the value of the value of the **dataFormat** property of that **JadeRestRequest**. For example, the following code fragment serializes the **Customer** object **theCustomer** into JSON format and adds it to the body of the **JadeRestRequest** object **myRequest**.

```
myRequest.dataFormat := JadeRestRequest.DataFormat_JSON;
myRequest.addObjectParam(theCustomer, Customer);
```

Alternatively, to serialize it into XML format, simply change the data format, as shown in the following code fragment.

```
myRequest.dataFormat := JadeRestRequest.DataFormat_XML;
myRequest.addObjectParam(theCustomer, Customer);
```

Form URL Data

You can use the **JadeRestRequest** class **addFormUrlData** method to add key/value pairs to the HTTP request body in the format required for the **application/x-www-form-urlencoded** HTTP content-type, as follows.

```
request.dataFormat := JadeRestRequest.DataFormat_FormUrlEncoding;
request.addFormUrlData("name", name);
request.addFormUrlData("status", status);
```

Multipart Form Data

You can use the **JadeRestRequest** class **addMultipartFormData** method to add content to the HTTP request body in the format required for the **multipart/form-data** HTTP content type; for example:

```
request.dataFormat := JadeRestRequest.DataFormat_MultipartFormData;
request.addMultipartFormData(
                "additionalMetadata",
                null,
                "application/json",
                additionalMetadata.String);
request.addMultipartFormData(
                "file",
                "file",
                "application/octet-stream",
                file.String);
```

The JadeRestResponse Class

The **JadeRestResponse** class represents a response returned from the server as the result of a specific request.

It contains information about the response; for example, the status code and data format for the request. It also contains the response body (data) returned from the server, as well as methods to deserialize it.

When making a REST request, you need to pass a **JadeRestResponse** as a parameter, which will be populated with the response information.

JadeRestResponse Class Methods

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

Method	Description
deserialize	Deserializes the response body back into objects
getDataWithoutMarkup	Standardizes primitive value response bodies by removing markup

Deserializing an Object Result

You can use the **JadeRestResponse** class **deserialize** method to deserialize a serialized object or collection.

You will need to provide two parameters to the method, which are the type of the object being deserialized and an **ObjectArray** into which to put the deserialized object and any references. The first entry in this array will be the deserialized object and then any referenced objects will be inserted after it; for example:

```
jadeRestResponse.deserialize(Customer, objs);
customer := objs.first.Customer;
```

Standardizing a Primitive Result

When receiving a primitive result from a REST service, you don't need to deserialize it.

The JSON and XML formats are used to serialize an object into a text representation, but primitive types already have a reasonable text representation. That said, it is common for REST services to put some amount of markup around a primitive type response or surround the result with quote characters.

You can use the **JadeRestResponse** class **getDataWithoutMarkup** method to remove any markup or quotes from a response body, leaving only the primitive type result. The method will always return a **String**. If the primitive is a different type, you will need to type-cast it to the appropriate type; for example:

```
intResult := jadeRestResponse.getDataWithoutMarkup().Integer;
```

Putting it all Together

Using the **JadeRestClient**, **JadeRestRequest**, **JadeRestResponse** classes, you can consume REST services from Jade.

The following sections contain code examples for requests of the four most-common HTTP verbs, and each will use the Swagger Petstore, a public demonstration REST API found at https://petstore.swagger.io/.

Note These examples assume you have classes that mirror the data model of the REST service (for example, a **Pet** class description). You can do this manually by inspecting the API documentation for the REST service you are consuming or by generating it automatically, which we will cover under "OpenAPI Imports", later in this white paper.

GET Method Example

A **GET** request is used for obtaining a resource.

It is not usual for any HTTP body to be required for a **GET** request, but the REST service will typically return the requested resource in the response body on a successful request.

```
getPet() : Pet;
constants
    Endpoint = "https://petstore.swagger.io/v2";
    Path = "/pet/{petId}";
vars
    client : JadeRestClient;
    response : JadeRestResponse;
    request : JadeRestRequest;
    objs : ObjectArray;
begin
    client := create JadeRestClient(Endpoint) transient;
    request := create JadeRestRequest(Path) transient;
    request.addURLSeg("petID", "1");
    create response transient;
    client.get(request, response);

    create objs transient;
    response.deserialize(Pet, objs);
    return objs.first.Pet;
epilog
    delete client;
    delete response;
    delete request;
```

```
    delete objs;
end;
```

POST Method Example

A **POST** request is used for adding a new resource to the REST service.

A **POST** request will usually require a serialized object to be included in the HTTP body.

```
postPet();
constants
    Endpoint = "https://petstore.swagger.io/v2";
    Path = "/pet";
vars
    client : JadeRestClient;
    response : JadeRestResponse;
    request : JadeRestRequest;
    pet : Pet;
begin
    client := create JadeRestClient(Endpoint) transient;
    request := create JadeRestRequest(Path) transient;

    create pet transient;
    pet.id := 314;
    pet.name := "Fluffy";
    pet.status := "available";
    request.dataFormat := request.DataFormat_JSON;
    request.addObjectParam(pet, Pet);

    create response transient;
    client.post(request, response);
    // We should get a 200 - Success.
    write response.statusCode;
epilog
    delete client;
    delete response;
    delete request;
    delete pet;
end;
```

PUT Method Example

A **PUT** request is used to replace an existing resource with a new one. It functions in a similar way to a **POST** request, except that an existing resource will be overwritten on the REST service.

```
putPet();
constants
    Endpoint = "https://petstore.swagger.io/v2";
    Path = "/pet";
vars
    client : JadeRestClient;
    response : JadeRestResponse;
    request : JadeRestRequest;
    pet : Pet;
begin
    client := create JadeRestClient(Endpoint) transient;
    request := create JadeRestRequest(Path) transient;
```

```
    create pet transient;
    pet.id := 314;
    pet.name := "Not Fluffy";
    pet.status := "available";
    request.dataFormat := request.DataFormat_JSON;
    request.addObjectParam(pet, Pet);

    create response transient;
    client.put(request, response);
    // We should get a 200 - Success.
    write response.statusCode;
epilog
    delete client;
    delete response;
    delete request;
    delete pet;
end;
```

DELETE Method Example

A **DELETE** request is used to delete an existing resource. It will not usually require any HTTP body for the request, nor will it return anything in the response body.

Note For the **DELETE** verb, the method you need to call is the **JadeRestClient** class **deleteResource** method rather than **delete**, as **delete** is a Jade reserved word.

```
deletePet();
constants
    Endpoint = "https://petstore.swagger.io/v2";
    Path = "/pet/{petId}";
vars
    client : JadeRestClient;
    response : JadeRestResponse;
    request : JadeRestRequest;
begin
    client := create JadeRestClient(Endpoint) transient;
    request := create JadeRestRequest(Path) transient;
    request.addURLSeg("petID", "314");
    create response transient;
    client.deleteResource(request, response);
    write response.statusCode;
epilog
    delete client;
    delete response;
    delete request;
end;
```

OpenAPI Imports

This section contains:

What is OpenAPI?

OpenAPI is a standardized format that describes REST APIs in a way that is readable by both humans and machines.

An OpenAPI specification fully describes a REST API including not only available endpoints and resources, the supported HTTP operations on those resources and the data model of the API, but also information such as licensing information, contact information, and terms of use.

From Jade 2020 and higher, you can import OpenAPI-compliant specifications into Jade, automatically generating proxy classes, the data model, and methods for all specified operations of the API. This allows you to use a REST API as if it were classes and methods of your own schema.

Importing an OpenAPI Specification

To show the import process for an OpenAPI specification, we use the specification for the public example REST API; that is, the Swagger Petstore.

The specification for the Swagger Petstore can be found at https://petstore.swagger.io/ or by making a **GET** REST request to https://petstore.swagger.io/v2/swagger.json.

The OpenAPI Import Wizard

You can use the OpenAPI Import Wizard to import OpenAPI Specifications.

To open the OpenAPI Import Wizard

1. In the Schema Browser, select the schema into which to import the OpenAPI specification.
2. From the Browse menu in the Class Browser, select **External Component Libraries** (Ctrl+Shift+R).
3. Select the **OpenAPI** sheet.
4. Right-click on the OpenAPI sheet to open the context menu, then select **Import**.

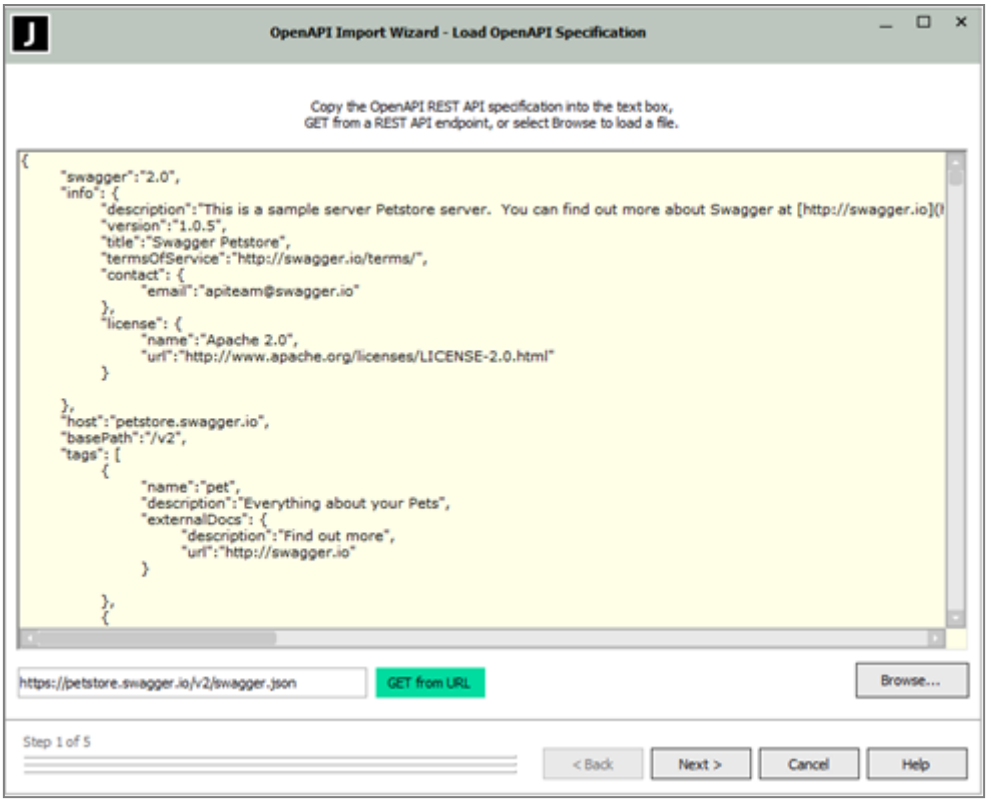
The OpenAPI Import Wizard is then displayed.

Loading an OpenAPI Specification

From the first step of the OpenAPI Import Wizard, you can load an OpenAPI specification one of the following ways.

- Copy and paste the specification into the main text box.
- Click the **Browse** button and select a file containing the specification.
- Enter a REST URL into the URL text box at the lower left of the form and then click the GET from URL button.

For example, enter https://petstore.swagger.io/v2/swagger.json in the URL text box and then click the **GET from URL** button to fetch the Swagger Petstore OpenAPI specification and display it in the main text box, as shown in the example in the following image.



When a valid OpenAPI specification is displayed in the main text box, click the **Next** button.

Naming the REST API

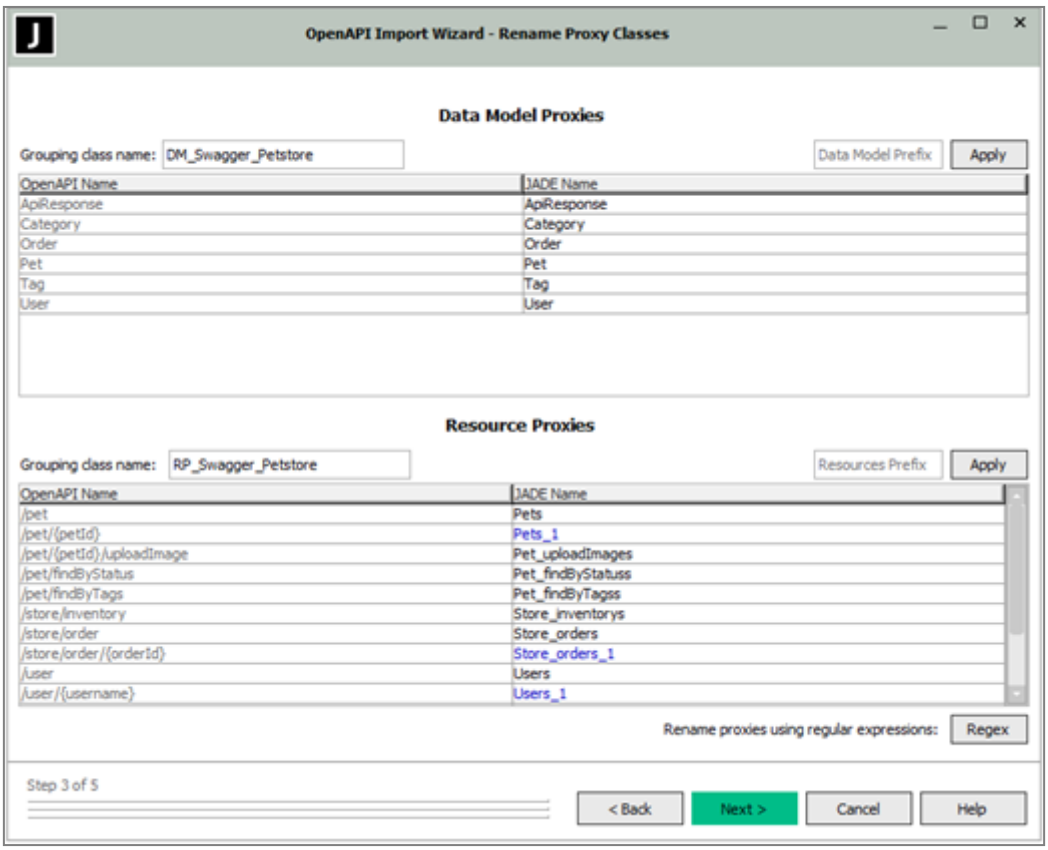
The REST API name is obtained automatically, by taking the title field of the specification and converting it to a valid Jade class name; that is, replacing invalid characters with underscore characters and capitalizing the first character. You can change it to any valid Jade class name if you do not want to use this default name.

When you are satisfied with the REST API name, click the **Next** button.

Renaming Proxy Classes

A Jade class is created for each data model class and resource in the specification. In addition, a grouping class is created as the superclass of the data model classes, and another as the superclass of the resource classes.

You can rename each of these classes on the **Rename Proxy Classes** sheet of the wizard, as shown in the example in the following image.



The classes have the following default names. For the:

- Grouping classes, the API name set in the previous step of the wizard is prefixed with **DM_** for the data model grouping class, and **RP_** for the resource proxy grouping class.
- Data model classes, the names are as close as possible to those in the specification, except the first character uppercase if required, any invalid characters are replaced by underscore characters, and any clashes are resolved by appending **_1** (or **_2**, **_3**, and so on, for multiple clashes).
- Resource classes, any parameters will be removed, and the name will be suffixed with s (resources are by convention described as plural). Then the same process as for the data model classes will be applied to ensure a valid Jade class name with no clashes.

You can rename any of these classes; for example, you can rename **Store_inventorys** to **Store_inventories**. Any class that has been renamed from its default value, including by use of **_1** (and so on) to avoid conflicts is colored blue.

As you rename classes, the names are automatically validated and if there are any conflicts, either within the generated classes or with an existing class of the schema, a validation error message is displayed and the **Next** button is disabled until all classes pass validation.

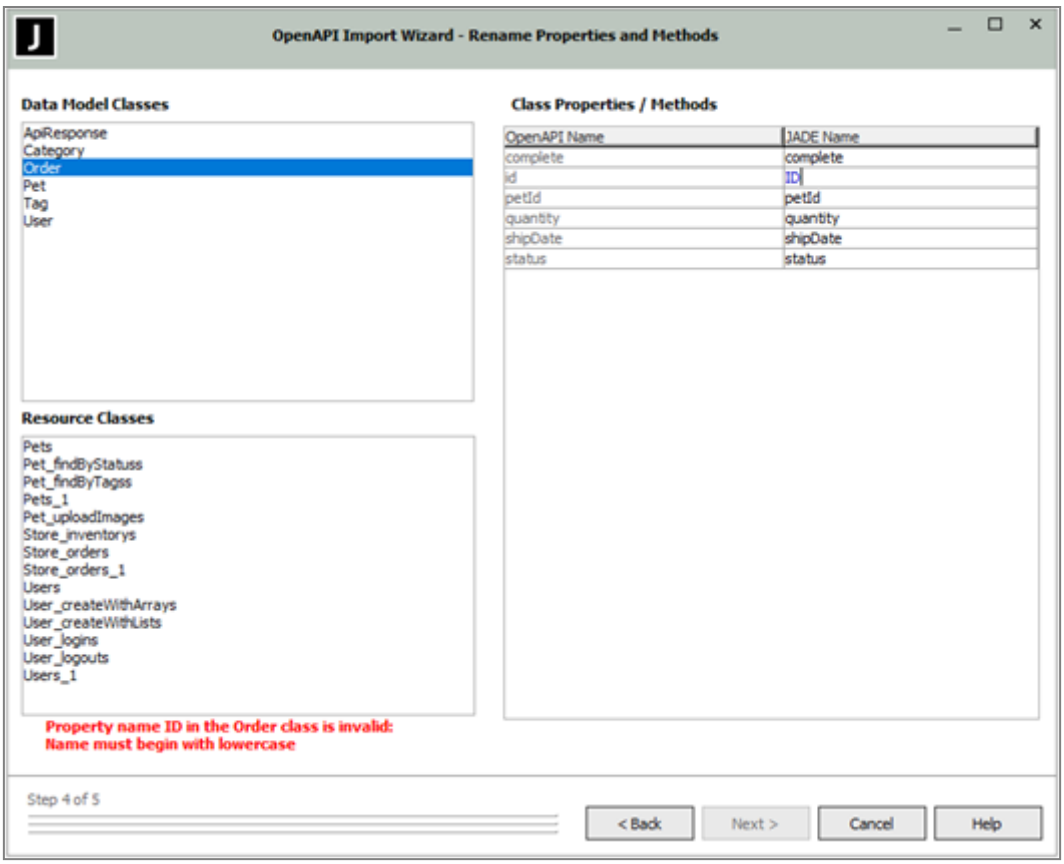
If you want to add a prefix to all data model classes or all resource proxy classes, enter the prefix in the **Data Model Prefix** or the **Resource Prefix** text box and then click the appropriate **Apply** button.

You can also use a regular expression (Regex) to modify the class names. Click on the **Regex** button to open the OpenAPI Wizard Regex dialog. If you want to test a Regex pattern before applying it to the class names, you can use one of the **Test** buttons on the dialog to apply the Regex pattern to the example text, which enables you to verify that it behaves as you expect before applying it to the class names.

When you are satisfied with the class names, click the **Next** button on the OpenAPI Wizard form.

Renaming Properties and Methods

In the **Rename Properties and Methods** step of the wizard, you can choose names for each of the properties of the data model classes, and each of the methods of the resource proxy classes, as shown in the example in the following image.

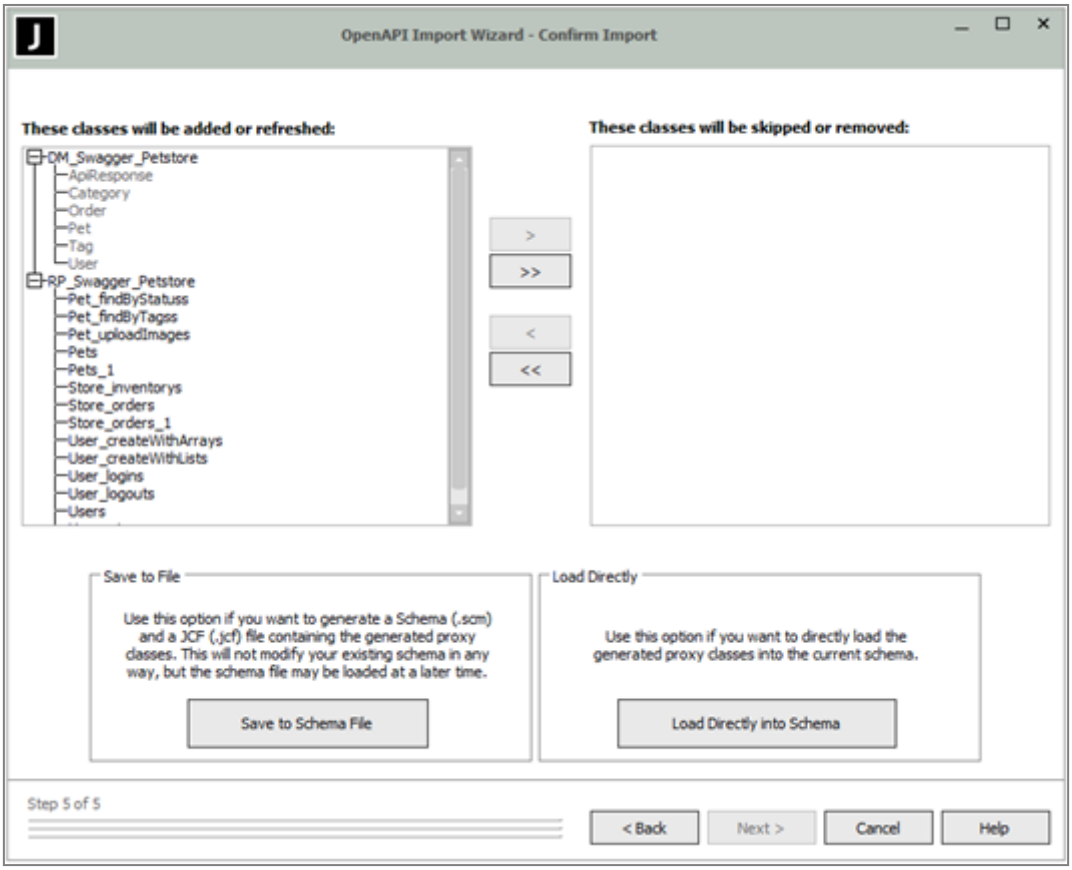


As in the previous step ("Renaming Proxy Classes"), any changed properties or methods are colored blue, and any invalid names or name clashes result in an error message being displayed.

When you are satisfied with the property and method names, click the **Next** button.

Excluding Resource Proxy Classes and Loading the API

The final step of the wizard, shown in the following image, enables you to remove any resource proxy classes that you do not want to import.



To exclude a single class, select it in the left-hand list and then click the **>** button. To exclude all of the resource proxy classes, click the **>>** button. You can use the **<** button to re-add a specific class or use the **<<** button to re-add all resource proxy classes.

Note Data model classes cannot be excluded, as resource proxy classes may depend on them.

When you are satisfied with the classes to import, you can click the **Save to Schema File** button to save the imported API to a schema (.**scm**) file or you can click the **Load Directly into Schema** button to load the imported API directly into the current schema.

When loading directly into the schema, the new classes are added to the *latest* version of the schema, with the schema being versioned, if necessary.

Using the Generated Proxy Classes

When the proxy classes have been generated into your schema, you can access the API as if it were your own classes. For example, you can perform a **POST** operation on a pet by simply instantiating the pet and passing it to the appropriate method of the appropriate resource proxy class, as shown in the following example.

```
exampleRest();
vars
    pet : Pet;
    api : Pets;
begin
    create pet transient;
    pet.name := "Fluffy";
    pet.status := "For Sale";
    pet.id := 314;

    create api transient;
    api.putPets(pet);
epilog
    delete pet;
    delete api;
end;
```