

1. REST API Developer Documentation	2
1.1 REST API Development	2
1.1.1 Guidelines for Atlassian REST API Design	3
1.1.1.1 Atlassian REST API Design Guidelines version 1	3
1.1.2 Overview of REST Implementation using the REST Plugin Module	10
1.1.3 REST Plugin Module	12
1.1.4 REST and os_authType	15
1.1.5 REST API Plugin Version Matrix	16
1.2 REST Plugin Release Notes	16
1.2.1 REST Plugin 2.2 Release Notes	16
1.2.2 REST Plugin 2.1 Release Notes	17
1.2.3 REST Plugin 2.0 Release Notes	17
1.2.4 REST Plugin 2.3 Release Notes	18
1.3 REST Glossary - Terms and Definitions	18
1.3.1 Idempotence or idempotent (Glossary Entry)	19
1.3.2 Representation (Glossary Entry)	19
1.3.3 Resource (Glossary Entry)	19
1.3.4 REST (Glossary Entry)	19

# REST API Developer Documentation

## Getting Started

If you would like to know more about REST in general, start with the [RESTwiki's](#) guide to [REST In Plain English](#). Then jump right in and try our REST resources. Refer to the application-specific REST API documentation, listed below. If you want to develop your own REST APIs for an Atlassian application, you will be interested in the [Atlassian REST plugin module](#), which is bundled with our applications.

## Application-Specific REST API Documentation

[Bamboo REST APIs](#)  
[Confluence REST APIs](#)  
[Crowd REST APIs](#)  
[FishEye/Crucible REST APIs](#)  
[JIRA REST APIs](#) (*Experimental and subject to change.*)

## API Development

### Overview of REST implementation

Read an overview of a REST implementation, using the Atlassian REST plugin module type.

### Atlassian Plugin SDK

Get started with developing an Atlassian plugin.

### REST plugin module type

Use the REST module type to create plugin points in Atlassian applications, by exposing services and data entities as REST APIs.

### Principles behind the Atlassian REST API design

Understand the guidelines followed by the Atlassian developers who are designing REST APIs for Atlassian applications.

## Tutorials

[Plugin Tutorial - Writing REST Services](#)

[Using the FishEye REST API to Write a Gadget to Monitor Recent Changes](#)

## Help

[Developer Network](#)

[Answers from the community](#)

[Mailing lists at my.atlassian.com](#)

[Feature requests and bug reports](#)

[Atlassian developer blog](#)

### Atlassian Developer Blog



[The road to HAMS 3.0 - Transaction boundaries](#)

[Make your plugin sizzle with new Plugin Exchange enhancements](#)

[Testing's In Session](#)

[AtlasCamp 2011](#)

[Let the Developer Party Begin: AtlasCamp 2011 Save the Date](#)

## REST API Development

- [Guidelines for Atlassian REST API Design](#)
- [Overview of REST Implementation using the REST Plugin Module](#)
- [REST Plugin Module](#)
- [REST and os\\_authType](#)
- [REST API Plugin Version Matrix](#)

# Guidelines for Atlassian REST API Design

The following documents provide guidelines to Atlassian developers who are designing [REST](#) APIs for Atlassian applications. Other developers, who are not Atlassian staff, may also find these documents interesting.

- [Atlassian REST API Design Guidelines version 1](#)

## RELATED TOPICS

[Plugin Tutorial - Writing REST Services](#)  
[Welcome to the Atlassian Developer Network](#)

## Atlassian REST API Design Guidelines version 1

This document provides guidelines to Atlassian developers who are designing [REST](#) APIs for Atlassian applications. However, we are publishing these design principles and guidelines for viewing by the wider community, for two reasons:

- If you are a developer/administrator who wants to interact with the REST API in one or more of the Atlassian applications, it will help you to know the principles behind our REST API design.
- We invite and welcome feedback on these design principles.



### Document version control

This page is under version control. Any significant changes to these guidelines will undergo review and approval and will be published under a new version number, as reflected in the page title.

### On this page:

- [Background to Atlassian REST APIs](#)
- [Goal of these Guidelines](#)
- [Using these Guidelines](#)
- [REST Resources](#)
  - [URIs for Resources](#)
    - [URI Structure](#)
    - [Standard Query Parameters in URIs](#)
  - [Entities](#)
    - [Representations \(Content Types\) of Entities](#)
    - [Hypertext Linking within an Entity](#)
    - [Entity ID](#)
    - [Version Control for Entities](#)
    - [Collections of Entities](#)
    - [Title Expansion for Entities](#)
- [Version Control for APIs](#)
  - [When to Change the Version](#)
    - [Changes That Don't Require a New Version](#)
    - [Change That Require a New Version](#)
- [Security](#)
  - [Authentication](#)
  - [Authorisation](#)
  - [XSRF](#)
- [Caching](#)
- [Concurrency](#)
- [Not Yet Covered in these Guidelines](#)
- [Appendix A: Response Codes](#)
- [Appendix B: Basic Data Types](#)
  - [Link](#)
  - [Status](#)

## Background to Atlassian REST APIs

Atlassian is currently working towards creating standardised [REST](#) APIs across all of our applications. Some of our applications already provide REST APIs, and some applications are providing new REST APIs or updating their existing APIs in an upcoming release. (Date of writing this paragraph: April 2009.)

Each application (Confluence, JIRA, Crowd, etc) will provide its own REST APIs, exposing the application-specific functionality. The goal is for these application-specific APIs to share common design standards, as described on this page.

For details of each application's APIs, please refer to the development hub in the documentation for the application concerned.

## Goal of these Guidelines

These guidelines are for Atlassian developers who are designing REST APIs for Atlassian applications. The goal is to keep REST API implementations as consistent as possible. These guidelines apply to all REST APIs, including:

- REST APIs that use Atlassian's Jersey-based [REST plugin module type](#).
- Other REST APIs not based on the plugin.

## Using these Guidelines

### Following or Moving Away from the Guidelines

These guidelines provide one way of doing things that might not always be the only way nor the best way. If you think that is the case, feel free to *move away from* the guidelines. You should be aware of potential repercussions such as security issues, consistency issues, etc.



We strongly recommend that someone who is familiar with these guidelines should review your REST API code.

### Examples used in the Guidelines

The examples used in these guidelines assume that we have a REST API supplied by an Atlassian plugin called the 'Unified Plugin Manager' plugin, 'UPM' or 'upm'. (This is a real plugin that is currently under development.) The UPM allows you to discover and manage the plugins installed on an Atlassian application.

## REST Resources

### URIs for Resources

#### URI Structure

Given a list of `foo` entities, the following URI scheme provides access to the list of `foo` entities and to a particular `foo`:

URI	Method	Notes
<code>/f<sub>oo</sub></code>	GET	This returns a list of the <code>f<sub>oo</sub></code> items. By default items in the list are a minimal representation of a <code>f<sub>oo</sub></code> entity. Note that we use the <b>singular</b> for the directory name.
<code>/f<sub>oo</sub>/<code>{key}</code></code>	GET	This returns the full content of the <code>f<sub>oo</sub></code> identified by the given <code>key</code> .

Sub-elements of a `foo` entity are made available as sub-resources of `/foo/{key}`.

#### Examples

For our example, let's take a plugin resource within the 'upm' REST API.

Use this URI to access a list of plugins:

```
http://host:port/context/rest/upm/1/plugin
```

Use this URI to access the plugin resource with a key of `a-plugin-key`:

```
http://host:port/context/rest/upm/1/plugin/a-plugin-key
```

Structure of the above URI:

- `host` and `port` define the host and port where the application lives.
- `context` is the servlet context of the application. For example, for Confluence this would typically be `confluence`.
- `rest` denotes the REST API.
- `upm` is the path declared in the REST module type in the plugin descriptor.
- `1` is the API version. See the section on API version control [below](#).

### Standard Query Parameters in URIs

Below is a list of standard query parameters. These names are reserved in our REST APIs and should be used according to the notes in the table below.

Query Parameter	Notes
<code>expand</code>	Used for title expansion. See section on title expansion <a href="#">below</a> .
<code>start-index</code>	An integer specifying the starting point when paging through a list of entities. Defaults to 0.
<code>max-results</code>	The maximum number of results when paging through a list of entities. No default is provided. We recommend that APIs should define their own default. Applications should also define a hard limit on the number of items that can be requested at the same time.
<code>jsonp-callback</code>	Used to define the name of the JavaScript function to be used as the <b>JSONP</b> callback.

## Entities

### Representations (Content Types) of Entities

By default, REST APIs must support multiple content types.

Representation	Requested via...	Notes
JSON	Requested via one of the following: <ul style="list-style-type: none"><li>• <code>application/json</code> in the HTTP Accept header</li><li>• <code>.json</code> extension</li></ul>	
XML	Requested via one of the following: <ul style="list-style-type: none"><li>• <code>application/xml</code> in the HTTP Accept header</li><li>• <code>.xml</code> extension</li></ul>	
JSONP	Requested via one of the following: <ul style="list-style-type: none"><li>• <code>application/json</code> in the HTTP Accept header and <code>jsonp-callback</code> query parameter</li><li>• <code>.json</code> extension and <code>jsonp-callback</code> query parameter.</li></ul>	Only available on <code>GET</code> . The returned content type <b>MUST</b> be <code>application/javascript</code> .

### Hypertext Linking within an Entity

Entities can have links to each other. This is represented by the `<link>` tag. The `<link>` tag supports the following attributes:

Attribute	Description
<code>href</code>	The URI of the entity linked to.
<code>rel</code>	The relationship between the <i>containing</i> entity and the entity linked to. Examples of possible values: <ul style="list-style-type: none"><li>• <code>self</code> — denotes that the URI points to the entity itself, see Entity ID <a href="#">below</a>.</li><li>• <code>edit</code> — denotes the URI used to update the entity,</li><li>• <code>delete</code> — denotes the URI used to delete the entity,</li><li>• <code>add</code> — denotes the URI used to create the entity.</li></ul>
<code>title</code>	Optional. A short description of the entity linked to.

Links to the URI that will perform operations on an entity have their `rel` attribute set to `edit`, `delete` or `add`. We recommend that entities provide these links as part of their content.

Links should follow some simple rules:

- Links use the same base URL independently of server-side implementation. So, if the REST API is available at `http://host:port/context/rest/api/`, then links to other resources should use this as their base URL.
- Whenever the user specifies an extension, such as `.xml`, `.json`, etc, links should have this same extension as far as possible.
- Links do not have query parameters.

#### Example

The following plugin entity has a `<link>` tag with a `rel` attribute identifying the entity's own ID via a URI pointing to itself:

```
<plugin key="a-plugin-key" enabled="true" expand="modules,info">
  <link rel="self" href="http://host:port/context/rest/upm/1/plugin/a-plugin-key" />
  <info name="A plugin"/>
  <modules size="2" expand="module"/>
</plugin>
```

### Entity ID

Every addressable entity should have a `<link>` tag with the attribute `rel="self"` pointing to its own URI. See the section on linking [above](#).

### Version Control for Entities

Entities SHOULD be served with an [ETag header](#).

*Read-only* entities and lists of entities may be served without an ETag header. Providing an ETag header for these resources will help with caches and conditional requests, but the header need not be included if calculating the ETag is too expensive or difficult.

The ETag for a resource MUST be the same regardless of the requested representation or title expansion state.

The server MUST treat [If-Match](#) and [If-None-Match](#) headers provided by clients as defined in [RFC 2616](#).

PUT or DELETE API requests SHOULD provide an [If-Match](#) header, and SHOULD react meaningfully to 412 (Precondition Failed) responses.

### Collections of Entities

Collections of entities should define the following attributes:

Attribute	Description
size	The total size of items available in the collection. This can be different from the actual number of items currently listed in the collection if the <code>start-index</code> and <code>max-result</code> query parameters have been used.
start-index	If used to <i>filter</i> the collection elements.
max-result	If used to <i>filter</i> the collection elements.

### Example

In the response below, the `<modules>` entity has a `size` attribute indicating that there are 2 plugin modules in the collection:

```
<plugin key="a-plugin-key" enabled="true" expand="modules,info">
  <link rel="self" href="http://host:port/context/rest/upm/1/plugin/a-plugin-key" />
  <info name="A plugin"/>
  <modules size="2" expand="module"/>
</plugin>
```

### Title Expansion for Entities

In order to be minimise network traffic and simplify some APIs from the client perspective, we recommend that APIs provide title expansion. This works by using the `expand` query parameter and setting its value to the last path element of the entity's schema.

The `expand` query parameter allows a comma-separated list of identifiers to expand. For example, the value `modules,info` requests the expansion of entities for which the `expand` identifier is `modules` and `info`.

You can use the dot notation to specify expansion of entities within another entity. For example `modules.module` would expand the plugin entity (because its `expand` identifier is `modules`) and the module entities within the plugin. See [example 4 below](#).

Expandable entities should be declared by parent entities in the form of an `expand` attribute. In [example 1](#), the plugin entity declares `modules` and `info` as being expandable. The attribute should not be confused with the query parameter which specifies which entities *are* expanded.

#### Example 1. Accessing a Resource without Title Expansion

For our example, let's take a plugin resource within the 'upm' REST API.

Use this URI to access the plugin resource without specifying title expansion:

<http://host:port/context/rest/upm/1/plugin/a-plugin-key>

The response will contain:

```
<plugin key="a-plugin-key" enabled="true" expand="modules,info">
  <link rel="self" href="http://host:port/context/rest/upm/1/plugin/a-plugin-key" />
  <info name="A plugin"/>
  <modules size="2" expand="module"/>
</plugin>
```

#### Example 2. Expanding the `info` Element

Use the following URI to expand the `info` element in our plugin resource:

<http://host:port/context/rest/upm/1/plugin/a-plugin-key?expand=info>.

Now the response will contain:

```
<plugin key="a-plugin-key" enabled="true" expand="modules,info">
  <link rel="self" href="http://host:port/context/rest/upm/1/plugin/a-plugin-key" />
  <info name="A plugin">
    <description>This is an awesome plugin</description>
    <version>1.1</version>
  </info>
  <modules size="2" expand="module" />
</plugin>
```

### Example 3. Expanding the Collection of Modules

Use this URI to expand the module collection in our plugin resource:

<http://host:port/context/rest/upm/1/plugin/a-plugin-key?expand=modules>

The response will contain:

```
<plugin key="a-plugin-key" enabled="true" expand="modules,info">
  <link rel="self" href="http://host:port/context/rest/upm/1/plugin/a-plugin-key" />
  <info name="A plugin" />
  <modules size="2" expand="module">
    <module key="module-key-1">
      <link rel="self"
href="http://host:port/context/rest/upm/1/plugin/a-plugin-key/module/module-key-1" />
    </module>
    <module key="module-key-2">
      <link rel="self"
href="http://host:port/context/rest/upm/1/plugin/a-plugin-key/module/module-key-2" />
    </module>
  </modules>
</plugin>
```

Note that the above URI does not expand each individual module.

### Example 4: Using the Dot Notation to Expand Entities within another Entity

Use the following URI to expand the modules inside the collection within our plugin resource:

<http://host:port/context/rest/upm/1/plugin/a-plugin-key?expand=modules.module>

The response will contain:

```
<plugin key="a-plugin-key" enabled="true" expand="modules,info">
  <link rel="self" href="http://host:port/context/rest/upm/1/plugin/a-plugin-key" />
  <info name="A plugin" />
  <modules size="2" expand="module">
    <module key="module-key-1">
      <link rel="self"
href="http://host:port/context/rest/upm/1/plugin/a-plugin-key/module/module-key-1" />
      <name>Module 1</name>
      <description>This is my first module</description>
    </module>
    <module key="module-key-2">
      <link rel="self"
href="http://host:port/context/rest/upm/1/plugin/a-plugin-key/module/module-key-2" />
      <name>Module 2</name>
      <description>This is my second module</description>
    </module>
  </modules>
</plugin>
```

### Using Index Values to Expand a Collection

You can also set indices to expand a given set of items in the collection. The index is 0 based:

- `modules[3]` will expand only the module at index 3 of the collection.
- `modules[1:3]` will expand modules ranging from index 1 to 3 (included). Note that `[ : 3]` and `[ 3 : ]` notations also work and the range goes respectively from the beginning of the collection and to the end of the collection.
- `modules[-1]` will expand the last module. Other negative index values also work and indices are counted from the end of the collection.

## Version Control for APIs

APIs must be subject to version control. The version number of an API appears in its URI. For example, use this URI structure to request version 1 of the 'upm' API:

```
http://host:port/context/rest/upm/1/...
```

When an API has multiple versions, we recommend:

- Two versions should be supported at the same time.
- If two or more versions are available, applications may provide a shortcut to the latest version using the `latest` key-word.

For example, if versions 1 and 2 of the 'upm' API are available, the following two URIs would point to the same resources:

- `http://host:port/context/rest/upm/latest/`
- `http://host:port/context/rest/upm/2/`

### When to Change the Version

A version number indicates a certain level of backwards-compatibility the client can expect, and as such, extra care should be taken to maintain this trust. The following lists which types of changes necessitate a new version and which don't:

#### Changes That Don't Require a New Version

- New resources
- New HTTP methods on existing resources
- New data formats
- New attributes or elements on existing data types

#### Change That Require a New Version

- Removed or renamed URIs
- Different data returned for same URI
- Removal of support for HTTP methods on existing URIs

## Security

### Authentication

Every REST API MUST at least accept [basic authentication](#).

Other authentication options are optional, such as [Trusted Apps](#), OS username and password as query parameters, or 'remember me' cookies.

By default, access to all resources (using any method) requires the client to be authenticated. Resources that should be available anonymously MUST be marked as such. The default implementation SHOULD use the `AnonymousAllowed` annotation.

### Authorisation

Authorisation is NOT handled by the REST APIs themselves. It is the responsibility of the service layer to make sure the authenticated client (user) has the correct permissions to access the given resource.

### XSRF

To prevent [XSRF](#) attacks, REST APIs MUST NOT accept POST requests with the `application/x-www-form-urlencoded` content type. We accept `multipart/form-data` content type as long as you use only file parameters. To POST, PUT requests with files and other parameters you SHOULD use the `multipart/mixed` content type.

The XSRF attack vector is the web browser. As long as GET requests are [idempotent](#), only POST operations can be exploited for XSRF attacks. However, these are restricted by the web browser to content types `application/x-www-form-urlencoded` and `multipart/form-data`. Therefore, by refusing POST requests that use `application/x-www-form-urlencoded` or `multipart/form-data`, we prevent XSRF attacks throughout our REST APIs.

**Exception:** When using the REST APIs to serve *normal* pages you will have to enable POST with `application/x-www-form-urlencoded`. In that case you should put in place some protection against XSRF such as using a form token that is valid for only one POST request.

### Caching

REST APIs SHOULD support conditional GET requests. This will allow the client to cache resource representations. For conditional GET requests to work, the client MUST send the [ETag](#) value when requesting resources using the [If-None-Match](#) request header. The ETag is the one provided by a previous request to that same URI. See the section on version control for entities [below](#).

Server implementations should acknowledge the `If-None-Match` header and check whether the response should be a 304 (Not Modified).



See the section on response codes [below](#).

## Concurrency

PUT or DELETE API requests SHOULD provide an `If-Match` header, and SHOULD react meaningfully to 412 (Precondition Failed) responses. See the section on response codes [below](#).

The ETag is the one provided by a previous GET request to that same URI. See the section on version control for entities [below](#).

## Not Yet Covered in these Guidelines

Some items worth discussing in the guidelines are currently out of scope:

- Common entities
- Batching
- Gzip
- OpenSearch
- Internationalisation (i18n)

## Appendix A: Response Codes

This list shows the common HTTP response codes and some brief guidelines on how to use them. For the complete list of HTTP response codes, please refer to [section 6 of RFC 2616](#).

Code	Name	Description	Notes
200	OK	Request was processed as expected.	<ul style="list-style-type: none"><li>• GET request returns a representation of the requested entity,</li><li>• The body of other requests will be a Status entity as described in the Status section of this document <a href="#">below</a>.</li></ul>
201	Created	Request created an entity.	<ul style="list-style-type: none"><li>• This cannot happen on GET or DELETE requests.</li><li>• This will happen on POST and may happen on PUT requests.</li><li>• The response should set the <code>Location</code> header to the URI for the created resource.</li><li>• The body of the response is a Status entity as described in the Status section of this document <a href="#">below</a>.</li></ul>
202	Accepted	The request has been acknowledged but cannot be processed <i>in real time</i> . For example, the request may have scheduled a job on the server.	<ul style="list-style-type: none"><li>• The response should set the <code>Location</code> header with the URI to the resource representing the pending action.</li><li>• The body of the response is a Status entity as described in the Status section of this document <a href="#">below</a>.</li></ul>
204	No Content		
301	Moved Permanently	The requested resource has been moved to another location (URI).	<ul style="list-style-type: none"><li>• The response should set the <code>Location</code> header to the URI of the new location of the resource.</li><li>• The body of the response is a Status entity as described in the Status_ section of this document <a href="#">below</a>.</li></ul>
304	Not Modified	The requested resource has not been modified. The client's <i>cached</i> representation is still valid.	<ul style="list-style-type: none"><li>• No body is allowed for these responses.</li></ul>
401	Unauthorized	Client is not authenticated or does not have sufficient permission to perform this request.	<ul style="list-style-type: none"><li>• The body of the response is a Status entity as described in the Status section of this document <a href="#">below</a>.</li></ul>
404	Not found	No resource was found at this location (URI).	<ul style="list-style-type: none"><li>• The body of the response is a Status entity as described in the Status section of this document <a href="#">below</a>.</li></ul>

412	Precondition Failed	The client specified some preconditions that are not valid.	<ul style="list-style-type: none"> <li>The body of the response is a Status entity as described in the Status section of this document <a href="#">below</a>.</li> </ul>
5xx	Server-Side Error	Any server-side error.	<ul style="list-style-type: none"> <li>These codes should not be set programmatically and are reserved for unexpected errors.</li> </ul>

## Appendix B: Basic Data Types

### Link

The link element is used for hyperlinking entities and as entity IDs:

```
<link title="This is my resource" rel="edit" href="http://host:port/context/rest/api/1/myresource" />
```

See the sections on [linking](#) and [entity IDs](#) above.

### Status

Any request which produces a status code with no body will have a body formatted like this:

```
<status>
  <plugin key="a-plugin-key" version="1.0" />
  <status-code>201</status-code>
  <sub-code>604</sub-code>
  <message>Some human readable message</message>
  <etag>233223</etag>
  <resources-created>
    <link rel="self" href="http:..." />
  </resources-created>
  <resources-updated />
</status>
```

Element	Description
plugin	Describes the plugin which provides the REST API. This element is present only if the REST API is provided by a plugin.
status-code	The actual HTTP code of the response. See the section on response codes <a href="#">above</a> .
sub-code	Another code that defines the error more specifically. It is up to REST API developers to define their various codes.
resources-created	This element is present when resources have been created. It consists of <a href="#">link</a> elements to the created resources.
resources-updated	This element is present when resources have been updated. It consists of <a href="#">link</a> elements to the updated resources.

#### RELATED TOPICS

[Plugin Tutorial - Writing REST services](#)

[\[Glossary - Terms and Definitions\]](#)

[REST Plugin Module](#)

[Overview of REST Implementation using the REST Plugin Module](#)

[Welcome to the Atlassian Developer Network](#)

## Overview of REST Implementation using the REST Plugin Module

This page gives an overview of a REST implementation, using the [Atlassian REST plugin module](#).

### Getting from the URL to the Java Objects

Given a URL like this:

```
http://myhost.com:port/myapp/rest/api-name/api-version/resource-name
```

We can break the URL into these parts:

1. `http://myhost.com:port`
2. `/myapp`
3. `/rest`
4. `/api-name/api-version`
5. `/resource-name`

Mapping each part of the URL:

1. The operating system directs the request to the application server (e.g. Tomcat) that handles the specified port.
2. Tomcat directs the request to the application `myapp`.
3. The application's `web.xml` deployment descriptor file maps the URLs to the relevant servlets. So in this case, it maps `/rest` to our REST servlet, which points to our [REST plugin module type](#).
4. Now the REST plugin module takes over. The relevant part of the URL (`api-name` and `api-version`) are defined as the path and version in the `atlassian-plugin.xml` file e.g.

```
<rest key="helloWorldRest" path="/helloworld" version="1.0">
  <description>Provides hello world services.</description>
</rest>
```

5. The final part of the URL mapping (`resource-name` and sub-elements) is done via annotations on the class, used to declare the URI path, the HTTP method and the media type. [Jersey](#) (based on [JAX-RS](#)) reads the `@Provider` and `@Path` annotations and maps them to classes and methods, so that we know which method is called for each REST resource and method. See the [Jersey documentation](#).

## Getting from the Java Objects to XML and JSON

[JAXB](#) converts the Java classes to XML or JSON and vice versa, making use of [JAXB annotations](#). (Available in Java 1.5 and later.)

For example a Java `User` object with JAXB annotations may look something like this:

```
import javax.xml.bind.annotation.*;

@XmlRootElement
public class User
{
    @XmlElement
    private String firstName;

    @XmlElement
    private String lastName;

    // This private constructor isn't used by any code, but JAXB requires any
    // representation class to have a no-args constructor.
    private User() { }


    public User(String firstName, String lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    ...
}
```

The XML response content for user John Smith would be:

```
<user>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
</user>
```

By default, on conversion the XML element name will be the same as the object name. Alternatively, you can add the XML element name to the annotation. Something like this: `@XmlRootElement(name="principal")`. There are specific annotations for arrays, etc. See [JAXB's documentation](#) for more details on this.

 We have no schema or DTD.

Jersey also handles JSON based on the same JAXB objects as in the example above. The JSON for the example would be:

```
{
  "firstName": "John",
  "lastName": "Smith"
}
```

A trivial REST service class with a single URL returning an instance of `User` would look like this:

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Path("/")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public class RestHelloWorldService {
    @GET
    @Path("users")
    public Response getUncompletedUsers() {
        return Response.ok(new User("Fred", "Bloggs")).build();
    }
}
```

#### Related Topics

[Plugin Tutorial - Writing REST Services](#)  
[REST Plugin Module](#)  
[Guidelines for Atlassian REST API Design](#)

## REST Plugin Module

You can use the REST plugin module to create plugin points easily in Atlassian applications, by exposing services and data entities as REST APIs. The Atlassian REST plugin module is bundled with our applications.

REST APIs provide access to resources via URI paths. To use a REST API, your plugin or script will make an HTTP request and parse the response. You can choose JSON or XML for the response format. Your methods will be the standard HTTP methods like GET, PUT, POST and DELETE. Because the REST API is based on open standards, you can use any web development language to access the API.



#### Plugin Framework 2 Only

The REST plugin module described below is available only for OSGi-based plugins using version 2.2 or later of the Atlassian Plugin Framework.

#### On this page:

- [Purpose of the REST Plugin Module](#)
- [Configuration](#)
  - [Attributes](#)
  - [Elements](#)
- [Example](#)
- [Notes on REST API and Plugin Development](#)
  - [JAX-RS and Jersey](#)
  - [Including the REST Plugin Module into your Project](#)
  - [Developing your REST API as an Atlassian Plugin](#)
  - [Accessing your REST Resources](#)
  - [How the REST Plugin Module Finds your Providers and Resources](#)
  - [Easy Way to Request JSON or XML Response Format](#)
  - [Working with JSON in Firefox](#)
  - [Differences in JSON and XML rendering of REST responses](#)
- [Help, I'm still stuck!](#)
- [References](#)




### Purpose of the REST Plugin Module

REST plugin modules enable you to expose services and data as REST resources.

### Configuration

The root element for the REST plugin module is `rest`. It allows the following attributes and child elements for configuration.

### Attributes

Name	Required	Description	Default
key		The identifier of the plugin module, i.e. the identifier of the REST module. This key must be unique within the plugin where it is defined. Sometimes you will need to uniquely identify a module. Do this with the <b>complete module key</b> . A module with key <code>fred</code> in a plugin with key <code>com.example.modules</code> will have a complete key of <code>com.example.modules:fred</code> .	N/A
path		The path to the REST API exposed by this module. For example, if set to <code>/foo</code> , the REST API will be available at <a href="http://localhost:8080/context/rest/foo/1.0">http://localhost:8080/context/rest/foo/1.0</a> , where <code>1.0</code> is the version of the REST API.	N/A
version		This is the version of the REST API. This is not the same thing as the plugin version. Different versions of the same API can be provided by different plugins. Version numbers follow the same pattern as OSGi versions, i.e. <code>major.minor.micro.qualifier</code> where <code>major</code> , <code>minor</code> and <code>micro</code> are integers.	N/A

### Elements

Name	Required	Description	Default
description		The description of the plugin module, i.e. the description of the REST module. The 'key' attribute can be specified to declare a localisation key for the value instead of text in the element body.	N/A
package		The package from which to start scanning for resources and providers. Can be specified multiple times. Defaults to scanning the whole plugin. <b>Since 2.0</b>	N/A
dispatcher		Determines when the filter is triggered. You can include multiple <code>dispatcher</code> elements. If this element is present, its content must be one of the following: <code>REQUEST</code> , <code>INCLUDE</code> , <code>FORWARD</code> , <code>ERROR</code> . Note: This element is only available in <a href="#">Plugin Framework 2.5</a> and later. If this element is not present, the filter will be fired on all conditions. (This is also the behaviour for Plugin Framework releases earlier than 2.5.)	Filter will be triggered on all conditions

### Example

Here is an example `atlassian-plugin.xml` file containing a single public component:

```
<atlassian-plugin name="Hello World" key="example.plugin.helloworld" plugins-version="2">
  <plugin-info>
    <description>A basic REST module test</description>
    <vendor name="Atlassian Software Systems" url="http://www.atlassian.com" />
    <version>1.0</version>
  </plugin-info>

  <rest key="helloWorldRest" path="/helloworld" version="1.0">
    <description>Provides hello world services.</description>
  </rest>
</atlassian-plugin>
```

## Notes on REST API and Plugin Development

### JAX-RS and Jersey

Here is some information to be aware of when developing a REST plugin module:

- The REST module is based on [JAX-RS](#). Specifically, it uses [Jersey](#), which is the JAX-RS reference implementation.
- The REST module is based on **version 1.0.2 of Jersey**.
- Jersey is configured to use all its default providers, but you should be able to change the configuration by adding your own providers in your plugin. For example, you may want to [enhance your JSON](#).

See [Overview of REST Implementation using the REST Plugin Module](#).

### Including the REST Plugin Module into your Project

You can include the REST plugin module as a Maven dependency from our [Maven repository](#).

### Developing your REST API as an Atlassian Plugin

To develop a REST API and deploy it into an Atlassian application, you will follow the same process as for any other Jersey application:

1. Develop your JAX-RS resources, using the annotations `@Path`, `@Provider`, etc.
2. Bundle your resource classes in your plugin JAR file, along with the plugin descriptor `atlassian-plugin.xml`.
3. Deploy your plugin to the application.

See [Developing your Plugin using the Atlassian Plugin SDK](#) and [Creating your Plugin Descriptor](#).

### Accessing your REST Resources

Your REST resources will be available at this URL:

```
http://host:port/context/rest/helloworld/1.0
```

- `host` and `port` define the host and port where the application lives.
- `context` is the servlet context of the application. For example, for Confluence this would typically be `confluence`.
- `helloworld` is the path declared in the REST module type in the plugin descriptor.
- `1.0` is the API version.

If `1.0` is the latest version of the *helloworld* API installed, this version will also be available at this URL:

```
http://host:port/context/rest/helloworld/latest
```

### How the REST Plugin Module Finds your Providers and Resources

The REST plugin module scans your plugin for classes annotated with the `@Provider` and `@Path` annotation. The `@Path` annotations can be simply declared on a method within a class and do not need to be present at the entity level.

For those not familiar with the JAX-RS specification, the `@Path` annotation can be declared at a package, class, or method level. Furthermore, their effects are cumulative. For example, if you define this at the package level:

```
@Path("/admin")
package myPackage;
```

then this at the class level:

```
package myPackage;


@Path("/myGroup")
public class MyGroup {...};
```


then this at the method level:

```
@Path("/myResource")
public void getResource() {...};
```

The final URL would be for the `helloWorld` plugin above:

```
http://host:port/context/rest/helloworld/1.0/admin/myGroup/myResource
```

 The REST plugin module will not scan a library bundled with your plugin, typically bundled in the `META-INF/lib` directory of the JAR. So make sure you put all providers and resources at the top level in your plugin JAR.

 Only one resource method can be bound to the root `"/"` path.

### Easy Way to Request JSON or XML Response Format

When using JAX-RS (and Jersey), the standard way to specify the content type of the response is to use the HTTP Accept header. While this is a good solution, it is not always convenient.

The REST plugin module allows you to use an extension to the resource name in the URL when requesting JSON or XML.

For example, let's assume I want to request JSON data for the resource at this address:

```
http://host:port/context/rest/helloworld/latest/myresource
```

I have two options:

- **Option 1:** Use the HTTP Accept header set to `application/json`.
- **Option 2:** Simply request the resource via this URL:

```
http://host:port/context/rest/helloworld/latest/myresource.json
```

If I want content of type `application/xml`, I will simply change the extension to `.xml`. Currently the REST plugin module supports only `application/json` and `application/xml`.

### Working with JSON in Firefox

A handy tool: The [JSONView](#) add-on for Firefox allows you to view JSON documents in the browser, with syntax highlighting. Here is a link to the [Firefox add-on page](#).

### Differences in JSON and XML rendering of REST responses

By default, JSON responses include **any** object fields which are explicitly annotated with a JAXB annotation, while XML responses include public fields and fields with public getters.

To get the same behaviour for JSON you need to either annotate each field with `@XmlElement` or `@XmlAttribute`, or annotate the class or package with `@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)`.

### Help, I'm still stuck!

Start with [DEVNET:General Development Tips] and then proceed to [DEVNET:Plugin Development Tips, FAQ and Troubleshooting]

### References

- [JAX-RS \(JSR311\) home page](#)
  - [1.0 Spec \(pdf\)](#)
  - [1.0 API](#)
- [Jersey home page](#)
  - [Wiki](#)
  - [1.0.2 API \(implements JAX-RS 1.0\)](#)
  - [Configuring JSON for RESTful Web Services in Jersey 1.0](#)

### RELATED TOPICS

[Plugin Tutorial - Writing REST Services](#)  
[Developing your Plugin using the Atlassian Plugin SDK](#)  
[Overview of REST Implementation using the REST Plugin Module](#)  
[Atlassian REST API Design Guidelines version 1](#)  
[Guidelines for Atlassian REST API Design](#)  
[Atlassian Plugin Framework Documentation](#)

## REST and os\_authType

This functionality is part of the Atlassian REST plugin 2.1.0 and Seraph 2.2 releases. Its availability depends on the Atlassian application upgrading to those versions or later.

Product	Available Since
JIRA	4.2

One common problem with REST API access to Atlassian applications is dealing with authentication. One way to deal with this is to pass the username and password on every request via the `os_username` and `os_password` URL query parameters. This has the obvious downside of putting that information in every single request where it is more vulnerable to sniffing and more likely to end up inadvertently in proxy and server logs.

To avoid that, some remote clients use cookies, just like a web browser does. Instead of passing a username and password with every request, you acquire login and acquire a cookie and then submit the cookie with each request. The downside to this approach is that cookies eventually expire. When this happens you are treated as an anonymous user. As a user, you've probably noticed this when, once in a while, you go to an Atlassian app and have to log in again. As a human, this situation is pretty easy to detect and fix. For something that is

programmatically interacting with an application it is much harder to detect! Imagine that you submit a query to JIRA and, instead of getting back 200 results, you get back only the 5 issues that anonymous users can see. Nothing obviously **failed** but you don't get the results you want.


We've improved this behaviour in Atlassian applications by changing how cookie expiration is treated under the **/rest** URLs. If you submit an expired cookie to a REST resource under the **/rest** URL you will receive a 401 error response instead of silently being treated as an anonymous user. The behaviour under all other parts of the system remains exactly the same. This change only affects **/rest** URLs.

















However, some applications may want to replicate this behaviour across the entirety of the system. For instance, if you are performing some kind of screen-scraping you might like to have this happen **everywhere**. You can trigger this behaviour via the **os\_authType** query parameter in your URL. **os\_authType** has been extended to support the following parameters and behaviour:


os_authType	behaviour
basic	the server will return a 401 error response and perform an HTTP Basic Authentication challenge if no username and password is specified
cookie	the server will return a 401 error response if a valid cookie is not provided in the request
any	if a username and password are not specified <b>and</b> there is not valid cookie, the server will return a 401 error response

## REST API Plugin Version Matrix

The matrix below shows the applications which include and support the REST API plugin. The applications are listed horizontally across the top and the REST plugin versions are listed vertically on the left.

- Version numbers next to a tick  show the **earliest** release of the application which supports the relevant REST plugin version.
- Version numbers in brackets show a future application release expected to support the relevant version of the REST plugin.

	Bamboo	Confluence	Crowd	Crucible	FishEye	JIRA
REST plugin 1.0	 Bamboo 2.4			 Crucible 2.3	 FishEye 2.3	 JIRA 4.0
REST plugin 1.1	 Bamboo 2.6	 Confluence 3.1	 Crowd 2.1			
REST plugin 2.0		 Confluence 3.3				
REST plugin 2.1						 JIRA 4.2
REST plugin 2.2	(Bamboo 3.0)	 Confluence 3.4	 Crowd 2.2	 Crucible 2.4	 FishEye 2.4	 JIRA 4.3
REST plugin 2.5		(Confluence 4.0)	 Crowd 2.3			 JIRA 4.4

 The REST API plugin is bundled with all application versions as shown in the above table. You do not need to install the plugin into any application.

### RELATED TOPICS

[REST API Developer Documentation](#)

## REST Plugin Release Notes

- [REST Plugin 2.2 Release Notes](#)
- [REST Plugin 2.1 Release Notes](#)
- [REST Plugin 2.0 Release Notes](#)
- [REST Plugin 2.3 Release Notes](#)

## REST Plugin 2.2 Release Notes

**13 September 2010**

With pleasure, Atlassian presents the **Atlassian REST Plugin 2.2**.

Highlights of this release:

- Secure administration sessions.** Atlassian REST 2.2, when paired with [SAL 2.2](#), provides support for secure administration sessions, the aptly-named 'WebSudo'. Confluence already supports [WebSudo](#): When an administrator who is logged into Confluence attempts to access an administration function, they are prompted to log in again. Eventually, all the Atlassian applications will support WebSudo sessions. How can you get WebSudo with REST? See the [SAL documentation](#).



- **Faster startup time.** Now some Spring files are bundled with the REST plugin, thus minimising the effort of the plugin framework to load it. In addition, the REST plugin no longer bundles its own copy of JAXB, significantly reducing the number of classes it needs to load.

### Complete List of Fixes in this Release

JIRA Issues (8 issues)			
Key	Summary	Priority	Status
REST-153	standardize dependency versioning		Resolved
REST-151	upgrade SAL to 2.2.0 beta9, PLUG to 2.6.0 beta2		Resolved
REST-149	create a release note for rest 2.2.0		Resolved
REST-147	Create own host component import file to speed up startup		Resolved
REST-145	Upgrade refapptest plugin to 4.2		Resolved
REST-144	Upgrade SAL to 2.2 and Refapp to 2.7		Resolved
REST-143	Add a REST interceptor for the new WebSudo annotations (SAL 2.2)		Resolved
REST-96	javax.xml.stream package imported from host application		Resolved

## REST Plugin 2.1 Release Notes

28 July 2010

With pleasure, Atlassian presents the **Atlassian REST Plugin 2.1**.

Atlassian REST 2.1, when paired with Seraph 2.2, provides more reliable use of REST resources by scripts and programs. The full explanation is a bit technical. Here's a developer's way of putting it into human-readable words:

*You know how after a while you get logged out of JIRA or Confluence. As a person you notice that pretty easily because the page just looks different. But a program isn't that smart. It doesn't know that it has suddenly been logged out and is now doing things as 'anonymous' and not 'jsmith'. The new REST + Seraph combination means they get an error message instead of just suddenly becoming 'anonymous' when using REST.*

Do you generate your REST API documentation using the Jersey [WADL](#)? With version 2.1 of the REST plugin, you can now have JSON examples as well as XML in your generated documentation. The plugin provides a new doclet based on `com.sun.jersey.wadl.resourcedoc.ResourceDoclet` that allows JSON examples. You will need to change your `pom.xml` to use this doclet.

This release also enables self-expansion of entities, along with other improvements described below.

### Complete List of Fixes in this Release

JIRA Issues (5 issues)			
Key	Summary	Priority	Status
REST-141	atlassian-rest-common doesn't export all packages		Resolved
REST-140	Enable self expanding functionality		Closed
REST-138	Framework doesn't handle inherited ListWrapper fields		Resolved
REST-136	add request attribute for /rest/ to force os_authType for seraph		Resolved
REST-134	Allow for JSON examples in generated REST documentation		Resolved

## REST Plugin 2.0 Release Notes

10 June 2010

With pleasure, Atlassian presents the **Atlassian REST Plugin 2.0**.

### Complete List of Fixes in this Release

JIRA Issues (16 issues)			
Key	Summary	Priority	Status

REST-132	Remove JAXB from plugin as it should be provided from the app		Resolved
REST-131	Fix proxy in JerseyRequestFactory		Resolved
REST-129	Bump Jackson version from 1.4.1 to 1.4.4		Closed
REST-128	Extract dependencies to speed up startup		Resolved
REST-127	Remove javax.activation from being bundled		Resolved
REST-126	Reloading plugins throws exception via the PDK install plugin despite the reload working fine		Resolved
REST-125	ThrowableExceptionHandler screws up WebApplicationExceptions		Resolved
REST-123	Use of private JacksonJsonProvider makes it difficult for plugins to use their own Jackson configuration		Resolved
REST-121	The ExtensionJerseyFilter#excludes init parameter name changed, breaking AG		Resolved
REST-120	Move ExtensionJerseyFilter into atlassian-rest-common module		Resolved
REST-119	Upgrade to plugins 2.5		Resolved
REST-118	Convert integration tests to use amps		Resolved
REST-116	Configuration of which packages to scan		Resolved
REST-115	JSR-303-based validation interceptor		Resolved
REST-114	rest-common should have properly scoped dependencies		Resolved
REST-86	Enable transactions for REST resources		Resolved

## REST Plugin 2.3 Release Notes

14 October 2010

With pleasure, Atlassian presents the **Atlassian REST Plugin 2.3**.

This is a maintenance release. Its primary purpose is compatibility with [SAL 2.3](#).

### Complete List of Fixes in this Release

JIRA Issues (2 issues)			
Key	Summary	Priority	Status
REST-154	Classloader order in ProxyUtils incorrect		Resolved
REST-155	Upgrade to SAL 2.3.0		Resolved

## REST Glossary - Terms and Definitions

Here is a list of all entries in the glossary, plus the first few lines of content. Click a link to see the full text for each entry.

- [Idempotence or idempotent \(Glossary Entry\)](#) — A request is idempotent when it does not matter if the request is sent multiple times. The second and subsequent times will have exactly the same effect as the first time.
- [Representation \(Glossary Entry\)](#) — A representation is the physical representation of a [REST](#) resource and should correspond to a standard media type. Examples of representations: XML, JSON, HTML, images, sound, movies, etc.
- [Resource \(Glossary Entry\)](#) — In the context of [REST](#) APIs, a resource is an item of information (entity) that can be named and

represented. Resources in your service should be identified using URIs. Examples of resources: A document; the current weather in Sydney; a plugin; a plugin module; a collection of plugin modules; etc.

- [REST \(Glossary Entry\)](#) — REST is the Representational State Transfer architectural style. In essence, REST is a set of rules (constraints) that an architecture should conform to. This is in contrast to an 'unconstrained architecture' where services are free to define their own ideosyncratic interfaces. The most important aspect of REST is a **uniform interface** between components, allowing them to communicate in a standard way. Requests use the standard HTTP methods. GET, PUT and DELETE requests can do only what is expected. The effect is that your services are accessible through standard tools, and it is safe for other services and utilities to use yours in ways you did not predict.

## RELATED TOPICS

- [REST API Development](#)
- [REST Plugin Release Notes](#)
- [REST Glossary - Terms and Definitions](#)

## Idempotence or idempotent (Glossary Entry)

A request is idempotent when it does not matter if the request is sent multiple times. The second and subsequent times will have exactly the same effect as the first time.

## RELATED TOPICS

[Guidelines for Atlassian REST API Design](#)

## Representation (Glossary Entry)

A representation is the physical representation of a [REST](#) resource and should correspond to a standard media type. Examples of representations: XML, JSON, HTML, images, sound, movies, etc.

## RELATED TOPICS

[Guidelines for Atlassian REST API Design](#)

## Resource (Glossary Entry)

In the context of [REST](#) APIs, a resource is an item of information (entity) that can be named and represented. Resources in your service should be identified using URIs. Examples of resources: A document; the current weather in Sydney; a plugin; a plugin module; a collection of plugin modules; etc.

## RELATED TOPICS

[Guidelines for Atlassian REST API Design](#)

## REST (Glossary Entry)

REST is the Representational State Transfer architectural style. In essence, REST is a set of rules (constraints) that an architecture should conform to. This is in contrast to an 'unconstrained architecture' where services are free to define their own ideosyncratic interfaces. The most important aspect of REST is a **uniform interface** between components, allowing them to communicate in a standard way. Requests use the standard HTTP methods. GET, PUT and DELETE requests can do only what is expected. The effect is that your services are accessible through standard tools, and it is safe for other services and utilities to use yours in ways you did not predict.

Some useful references:

- Roy Thomas Fielding's dissertation on [Architectural Styles and the Design of Network-based Software Architectures](#)
- [Wikipedia](#)
- [REST wiki](#)
- [REST in plain English](#)
- [A blog post by Dare Obasanjo 17/8/08](#)
- [A blog post on how to create a REST protocol](#)
- [A blog post on how not to create a REST API](#)

## RELATED TOPICS

[Guidelines for Atlassian REST API Design](#)