Redmine API

Redmine exposes some of its data through a REST API. This API provides access and basic CRUD operations (create, update, delete) for the resources described below. The API supports both XML and JSON formats.

API Description

Resource	Status	Notes	Availability
Issues	Stable		1.0
Projects	Stable		1.0
Project Memberships	Alpha		1.4
Users	Stable		1.1
Time Entries	Stable		1.1
News	Prototype	Prototype implementation for `index` only	1.1
Issue Relations	Alpha		1.3
Versions	Alpha		1.3
Wiki Pages	Alpha		2.2
Queries	Alpha		1.3
Attachments	Beta	Adding attachments via the API added in 1.4	1.3
Issue Statuses	Alpha	Provides the list of all statuses	1.3
Trackers	Alpha	Provides the list of all trackers	1.3
Enumerations	Alpha	Provides the list of issue priorities and time tracking activities	2.2
Issue Categories	Alpha		1.3
Roles	Alpha		1.4

Groups	Alpha		2.1
Custom Fields	Alpha		2.4

Status legend:

- Stable - feature complete, no major changes planned
- Beta - usable for integrations with some bugs or missing minor functionality
- Alpha - major functionality in place, needs feedback from API users and integrators
- Prototype - very rough implementation, possible major breaking changes mid-version. **Not recommended for integration**
- Planned - planned in a future version, depending on developer availability

General topics

Specify `Content-Type` on `POST`/`PUT` requests

When creating or updating a remote element, the `Content-Type` of the request **MUST** be specified even if the remote URL is suffixed accordingly (e.g. `POST ../issues.json`):

- for JSON content, it must be set to `Content-Type: application/json`.
- for XML content, to `Content-Type: application/xml`.

Authentication

Most of the time, the API requires authentication. To enable the API-style authentication, you have to check **Enable REST API** in Administration -> Settings -> Authentication. Then, authentication can be done in 2 different ways:

- using your regular login/password via HTTP Basic authentication.
- using your API key which is a handy way to avoid putting a password in a script. The API key may be attached to each request in one of the following way:
  - passed in as a "key" parameter
  - passed in as a username with a random password via HTTP Basic authentication
  - passed in as a "X-Redmine-API-Key" HTTP header (added in Redmine 1.1.0)

You can find your API key on your account page (/my/account) when logged in, on the right-hand pane of the default layout.

User Impersonation

As of Redmine 2.2.0, you can impersonate user through the REST API by setting the `X-Redmine-Switch-User` header of your API request. It must be set to a user login (eg. `X-Redmine-Switch-User: jsmith`). This only works when using the API with an administrator account, this header will be ignored when using the API with a regular user account.

If the login specified with the `X-Redmine-Switch-User` header does not exist or is not active, you will receive a 412 error response.

Collection resources and pagination

The response to a GET request on a collection ressources (eg. `/issues.xml`, `/users.xml`) generally won't return all the objects available in your database. Redmine 1.1.0 introduces a common way to query such ressources using the following parameters:

- `offset`: the offset of the first object to retrieve
- `limit`: the number of items to be present in the response (default is 25, maximum is 100)

Examples:

```
GET /issues.xml
=> returns the 25 first issues

GET /issues.xml?limit=100
=> returns the 100 first issues

GET /issues.xml?offset=30&limit=10
=> returns 10 issues from the 30th
```

Responses to GET requests on collection ressources provide information about the total object count available in Redmine and the offset/limit used for the response. Examples:

```
GET /issues.xml

<issues type="array" total_count="2595" limit="25" offset="0">
  ...
</issues>
GET /issues.json

{ "issues":[...], "total_count":2595, "limit":25, "offset":0 }
```

Note: if you're using a REST client that does not support such top level attributes (total_count, limit, offset), you can set the `nometa` parameter or `X-Redmine-Nometa` HTTP header to 1 to get responses without them. Example:

```
GET /issues.xml?nometa=1

<issues type="array">
  ...
</issues>
```

Fetching associated data

Since of 1.1.0, you have to explicitly specify the associations you want to be included in the query result by appending the `include` parameter to the query url :

Example:

To retrieve issue journals with its description:

```
GET /issues/296.xml?include=journals

<issue>
  <id>296</id>
  ...
  <journals type="array">
  ...
  </journals>
</issue>
```

You can also load multiple associations using a coma separated list of items.

Example:

```
GET /issues/296.xml?include=journals,changesets

<issue>
  <id>296</id>
  ...
  <journals type="array">
  ...
  </journals>
  <changesets type="array">
  ...
  </changesets>
</issue>
```

Working with custom fields

Most of the Redmine objects support custom fields. Their values can be found in the `custom_fields` attributes.

XML Example:

```
GET /issues/296.xml      # an issue with 2 custom fields
```

```
<issue>
  <id>296</id>
  ...
  <custom_fields type="array">
    <custom_field name="Affected version" id="1">
      <value>1.0.1</value>
    </custom_field>
    <custom_field name="Resolution" id="2">
      <value>Fixed</value>
    </custom_field>
  </custom_fields>
</issue>
```

JSON Example:

```
GET /issues/296.json      # an issue with 2 custom fields

{"issue":
  {
    "id":8471,
    ...
    "custom_fields":
      [
        {"value":"1.0.1","name":"Affected version","id":1},
        {"value":"Fixed","name":"Resolution","id":2}
      ]
  }
}
```

You can also set/change the values of the custom fields when creating/updating an object using the same syntax (except that the custom field name is not required).

XML Example:

```
PUT /issues/296.xml

<issue>
  <subject>Updating custom fields of an issue</subject>
  ...
  <custom_fields type="array">
```

```
    <custom_field id="1">
      <value>1.0.2</value>
    </custom_field>
    <custom_field id="2">
      <value>Invalid</value>
    </custom_field>
  </custom_fields>
</issue>
```

Note: the `type="array"` attribute on `custom_fields` XML tag is strictly required.

JSON Example:

```
PUT /issues/296.json

{"issue":
  {
    "subject":"Updating custom fields of an issue",
    ...
    "custom_fields":
      [
        {"value":"1.0.2","id":1},
        {"value":"Invalid","id":2}
      ]
  }
}
```

Attaching files

Support for adding attachments through the REST API is added in Redmine 1.4.0.

First, you need to upload your file with a POST request to `/uploads.xml` (or `/uploads.json`). The request body should be the content of the file you want to attach and the `Content-Type` header must be set to `application/octet-stream` (otherwise you'll get a `406 Not Acceptable` response). If the upload succeeds, you get a 201 response that contains a token for your uploaded file.

```
POST /uploads.xml
Content-Type: application/octet-stream
...
(request body is the file content)
```

```
# 201 response
<upload>
  <token>7167.ed1ccdb093229ca1bd0b043618d88743</token>
</upload>
```

Then you can use this token to attach your uploaded file to a new or an existing issue.

```
POST /issues.xml
<issue>
  <project_id>1</project_id>
  <subject>Creating an issue with a uploaded file</subject>
  <uploads type="array">
    <upload>
      <token>7167.ed1ccdb093229ca1bd0b043618d88743</token>
      <filename>image.png</filename>
      <description>An optional description here</description>
      <content_type>image/png</content_type>
    </upload>
  </uploads>
</issue>
```

If you try to upload a file that exceeds the maximum size allowed, you get a 422 response:

```
POST /uploads.xml
Content-Type: application/octet-stream
...
(request body larger than the maximum size allowed)


# 422 response
<errors>
  <error>This file cannot be uploaded because it exceeds the maximum allowed
file size (1024000)</error>
</errors>
```

Validation errors

When trying to create or update an object with invalid or missing attribute parameters, you will get a `422 Unprocessable Entity` response. That means that the object could not be created or updated. In such cases, the response body contains the corresponding error messages:

XML Example:

```
# Request with invalid or missing attributes
POST /users.xml
<user>
  <login>john</login>
  <lastname>Smith</lastname>
  <mail>john</mail>
</uer>

# 422 response with the error messages in its body
<errors type="array">
  <error>First name can't be blank</error>
  <error>Email is invalid</error>
</errors>
```

JSON Example:

```
# Request with invalid or missing attributes
POST /users.json
{
  "user":{
    "login":"john",
    "lastname":"Smith",
    "mail":"john"
  }
}

# 422 response with the error messages in its body
{
  "errors":[
    "First name can't be blank",
    "Email is invalid"
  ]
}
```

JSONP Support

Redmine 2.1.0+ API supports JSONP to request data from a Redmine server in a different domain (say, with JQuery). The callback can be passed using the `callback` or `jsonp` parameter. As of Redmine 2.3.0, JSONP support is optional and disabled by default, you can enable it by checking **Enable JSONP support** in Administration -> Settings -> Authentication.

Example:

```
GET /issues.json?callback=myHandler



myHandler({"issues":[ ... ]})
```

API Usage in various languages/tools

- Ruby
- PHP
- Python
- Java
- cURL
- Drupal Redmine API module, 2.x branch
- .NET
- Delphi

API Change history

This section lists changes to the existing API features only. New features of the API are listed in the API Description.

2012-01-29: Multiselect custom fields (r8721, 1.4.0)

Custom fields with multiple values are now supported in Redmine and may be found in API responses. These custom fields have a `multiple=true attribute` and their `value` attribute is an array.

Example:

```
GET /issues/296.json


{"issue":
  {
    "id":8471,
    ...
    "custom_fields":
      [
        {"value":["1.0.1","1.0.2"],"multiple":true,"name":"Affected
version","id":1},
        {"value":"Fixed","name":"Resolution","id":2}
      ]
  }
}
```

Issues

Listing issues

```
GET /issues.[format]
```

Returns a paginated list of issues. By default, it returns open issues only.

> Parameters:

- `offset`: skip this number of issues in response (optional)
- `limit`: number of issues per page (optional)
- `sort`: column to sort with. Append `:desc` to invert the order.

Optional filters:

- `project_id`: get issues from the project with the given id, where id is either project id or project identifier
- `subproject_id`: get issues from the subproject with the given id. You can use `project_id=XXX&subproject_id=!*` to get only the issues of a given project and none of its subprojects.
- `tracker_id`: get issues from the tracker with the given id
- `status_id`: get issues with the given status id only. Possible values: `open`, `closed`, `*` to get open and closed issues, status id
- `assigned_to_id`: get issues which are assigned to the given user id. `me` can be used instead an ID to fetch all issues from the logged in user (via API key or HTTP auth)
- `cf_x`: get issues with the given value for custom field with an ID of `x`. (Custom field must have 'used as a filter' checked.)
- ...

NB: operators containing ">", "<" or "=" should be hex-encoded so they're parsed correctly. Most evolved API clients will do that for you by default, but for the sake of clarity the following examples have been written with no such magic feature in mind.

> Examples:

```
GET /issues.xml

GET /issues.xml?project_id=2
GET /issues.xml?project_id=2&tracker_id=1
GET /issues.xml?assigned_to_id=6
GET /issues.xml?assigned_to_id=me
GET /issues.xml?status_id=closed
GET /issues.xml?status_id=*
GET /issues.xml?cf_1=abcdef
GET /issues.xml?sort=category:desc,updated_on

Paging example:
GET /issues.xml?offset=0&limit=100
GET /issues.xml?offset=100&limit=100

To fetch issues for a date range (uncrypted filter is "><2012-03-01|2012-03-
07") :
GET /issues.xml?created_on=%3E%3C2012-03-01|2012-03-07

To fetch issues created after a certain date (uncrypted filter is ">=2012-03-
01") :
GET /issues.xml?created_on=%3E%3D2012-03-01

Or before a certain date (uncrypted filter is "<= 2012-03-07") :
GET /issues.xml?created_on=%3C%3D2012-03-07

To fetch issues created after a certain timestamp (uncrypted filter is
">=2014-01-02T08:12:32Z") :
GET /issues.xml?created_on=%3E%3D2014-01-02T08:12:32Z

To fetch issues updated after a certain timestamp (uncrypted filter is
">=2014-01-02T08:12:32Z") :
GET /issues.xml?updated_on=%3E%3D2014-01-02T08:12:32Z
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<issues type="array" count="1640">
  <issue>
    <id>4326</id>
    <project name="Redmine" id="1"/>
    <tracker name="Feature" id="2"/>
    <status name="New" id="1"/>
    <priority name="Normal" id="4"/>
    <author name="John Smith" id="10106"/>
    <category name="Email notifications" id="9"/>
    <subject>
      Aggregate Multiple Issue Changes for Email Notifications
    </subject>
    <description>
      This is not to be confused with another useful proposed feature that
      would do digest emails for notifications.
    </description>
    <start_date>2009-12-03</start_date>
    <due_date></due_date>
    <done_ratio>0</done_ratio>
    <estimated_hours></estimated_hours>
    <custom_fields>
      <custom_field name="Resolution" id="2">Duplicate</custom_field>
      <custom_field name="Texte" id="5">Test</custom_field>
      <custom_field name="Boolean" id="6">1</custom_field>
      <custom_field name="Date" id="7">2010-01-12</custom_field>
    </custom_fields>
    <created_on>Thu Dec 03 15:02:12 +0100 2009</created_on>
    <updated_on>Sun Jan 03 12:08:41 +0100 2010</updated_on>
  </issue>
  <issue>
    <id>4325</id>
    ...
  </issue>
</issues>
```

Showing an issue

```
GET /issues/[id].[format]
```

> Parameters:

- `include`: fetch associated data (optional, use comma to fetch multiple associations). Possible values:
  - `children`
  - `attachments`
  - `relations`
  - `changesets`
  - `journals` - See Issue journals for more information.

    - watchers - Since 2.3.0

> Examples:

```
GET /issues/2.xml
GET /issues/2.json

GET /issues/2.xml
GET /issues/2.xml?include=attachments
GET /issues/2.xml?include=attachments,journals
```

Creating an issue

```
POST /issues.[format]
```

> Parameters:

- issue - A hash of the issue attributes:
    - project_id
    - tracker_id
    - status_id
    - priority_id
    - subject
    - description
    - category_id
    - fixed_version_id - ID of the Target Versions (previously called 'Fixed Version' and still referred to as such in the API)
    - assigned_to_id - ID of the user to assign the issue to (currently no mechanism to assign by name)
    - parent_issue_id - ID of the parent issue
    - custom_fields - See Custom fields
    - watcher_user_ids - Array of user ids to add as watchers (since 2.3.0)

Attachments can be added when you create an issue, see Attaching files.

> Examples:

```
POST /issues.xml
<?xml version="1.0"?>
<issue>
  <project_id>1</project_id>
  <subject>Example</subject>
  <priority_id>4</priority_id>
</issue>
POST /issues.json
{
  "issue": {
    "project_id": 1,
    "subject": "Example",
    "priority_id": 4
  }
}
```

Updating an issue

```
PUT /issues/[id].[format]
```

> Parameters:

- `issue` - A hash of the issue attributes
    - `project_id`
    - `tracker_id`
    - `status_id`
    - `subject`
    - ...
    - `notes` - Comments about the update

Attachments can be added when you update an issue, see Attaching files.

> Examples:

```
PUT /issues/[id].xml
<?xml version="1.0"?>
<issue>
  <subject>Subject changed</subject>
  <notes>The subject was changed</notes>
</issue>
PUT /issues/[id].json
{
  "issue": {
    "subject": "Subject changed",
    "notes": "The subject was changed"
  }
}
```

Deleting an issue

```
DELETE /issues/[id].[format]
```

Adding a watcher

Added in 2.3.0

```
POST /issues/[id]/watchers.[format]
```

> Parameters:

- `user_id` (required): id of the user to add as a watcher

Removing a watcher

Added in 2.3.0

```
DELETE /issues/[id]/watchers/[user_id].[format]
```

> Parameters: *none*

Projects

Listing projects

```
GET /projects.xml
```

Returns all projects (all public projects and private projects where user have access to)

> Parameters:

- `include`: fetch associated data (optional). Possible values: trackers, issue_categories, enabled_modules (since 2.6.0). Values should be separated by a comma ",".

> Response:

```
<projects type="array">
  <project>
    <id>1</id>
    <name>Redmine</name>
    <identifier>redmine</identifier>
    <description>
      Redmine is a flexible project management web application written using
Ruby on Rails framework.
    </description>
    <created_on>Sat Sep 29 12:03:04 +0200 2007</created_on>
    <updated_on>Sun Mar 15 12:35:11 +0100 2009</updated_on>
    <is_public>true</is_public>
  </project>
  <project>
    <id>2</id>
    ...
  </project>
```

Notes:

- `is_public` is exposed since 2.6.0

Showing a project

```
GET /projects/[id].xml
```

Returns the project of given id or identifier.

Parameters:

- `include`: fetch associated data (optional). Possible values: trackers, issue_categories, enabled_modules (since 2.6.0). Values should be separated by a comma ",".

Examples:

```
GET /projects/12.xml
GET /projects/12.xml?include=trackers
GET /projects/12.xml?include=trackers,issue_categories
GET /projects/12.xml?include=enabled_modules
GET /projects/redmine.xml
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<project id="1">
  <name>Redmine</name>
  <identifier>redmine</identifier>
  <description>
    Redmine is a flexible project management web application written using
Ruby on Rails framework.
  </description>
  <homepage></homepage>
  <created_on>Sat Sep 29 12:03:04 +0200 2007</created_on>
  <updated_on>Sun Mar 15 12:35:11 +0100 2009</updated_on>
  <is_public>true</is_public>
</project>
```

Notes:

- `is_public` is exposed since 2.6.0

Creating a project

```
POST /projects.xml
```

Creates a the project.

Parameters:

- `project` (required): a hash of the project attributes, including:
  - `name` (required): the project name
  - `identifier` (required): the project identifier
  - `description`

Response:

- `201 Created`: project was created
- `422 Unprocessable Entity`: project was not created due to validation failures (response body contains the error messages)

Updating a project

```
PUT /projects/[id].xml
```

Updates the project of given id or identifier.

Deleting a project¶

```
DELETE /projects/[id].xml
```

Deletes the project of given id or identifier.

Limitations:

A POST request on Redmine 1.0.1-2 (debian stable) does not work using the API key, but does work with a login/passw authentication
http://www.redmine.org/issues/12104

Project Memberships

/projects/:project_id/memberships.:format

GET

Returns a paginated list of the project memberships. `:project_id` can be either the project numerical id or the project identifier.

> Examples:

```
GET /projects/1/memberships.xml

GET /projects/redmine/memberships.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<memberships type="array" limit="25" offset="0" total_count="3">
  <membership>
    <id>1</id>
    <project name="Redmine" id="1"/>
    <user name="David Robert" id="17"/>
    <roles type="array">
      <role name="Manager" id="1"/>
    </roles>
  </membership>
  <membership>
    <id>3</id>
    <project name="Redmine" id="1"/>
    <group name="Contributors" id="24"/>
    <roles type="array">
      <role name="Contributor" id="3"/>
    </roles>
  </membership>
  <membership>
    <id>4</id>
    <project name="Redmine" id="1"/>
    <user name="John Smith" id="27"/>
    <roles type="array">
      <role name="Developer" id="2" />
      <role name="Contributor" id="3" inherited="true" />
    </roles>
  </membership>
</memberships>
```

Notes:

- The membership owner can be either a user or a group (Groups API is added in Redmine 2.1)
- In the above example, the `inherited="true"` attribute on the last role means that this role was inherited from a group (eg. Jonh Smith belongs to the Contributors group and this group was added as a project member). John Smith's membership can not be deleted without deleting the group membership first.
- The memberships of a given user can be retrieved from the Users API.

POST

Adds a project member.

Parameters:

- `membership` (required): a hash of the membership attributes, including:
  - `user_id` (required): the numerical id of the user or group
  - `role_ids` (required): an array of roles numerical ids

Example:

```
POST /projects/redmine/memberships.xml


<membership>
  <user_id>27</user_id>
  <role_ids type="array">
    <role_id>2</role_id>
  </role_ids>
</membership>
```

JSON

```
{
  "membership":
  {
    "user_id": 27,
    "role_ids": [ 2 ]
  }
```

```
}
```

> Response:

- `201 Created`: membership was created
- `422 Unprocessable Entity`: membership was not created due to validation failures (response body contains the error messages)

/memberships/:id.:format

GET

Returns the membership of given `:id`.

> Examples:

```
GET /memberships/1.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<membership>
  <id>1</id>
  <project name="Redmine" id="1"/>
  <user name="David Robert" id="17"/>
  <roles type="array">
    <role name="Developer" id="2"/>
    <role name="Manager" id="1"/>
  </roles>
</membership>
```

PUT

Updates the membership of given :id. Only the roles can be updated, the project and the user of a membership are read-only.

> Parameters:

- `membership` (required): a hash of the membership attributes, including:
    - `role_ids` (required): an array of roles numerical ids

> Example:

```
PUT /memberships/2.xml


<membership>
  <role_ids type="array">
    <role_id>3</role_id>
    <role_id>4</role_id>
  </role_ids>
</membership>
```

> Response:

- `200 OK`: membership was updated
- `422 Unprocessable Entity`: membership was not updated due to validation failures (response body contains the error messages)

DELETE

Deletes a memberships.

Memberships inherited from a group membership can not be deleted. You must delete the group membership.

> Parameters:

none

> Example:

```
DELETE /memberships/2.xml
```

> Response:

- `200 OK`: membership was deleted
- `422 Unprocessable Entity`: membership was not deleted

Users

- Users
    - /users.:format
        - GET

/users.:format

GET

Returns a list of users.

Example:

```
GET /users.xml
```

Optional filters:

- `status`: get only users with the given status. See app/models/principal.rb for a list of available statuses. Default is 1 (active users)
- `name`: filter users on their login, firstname, lastname and mail ; if the pattern contains a space, it will also return users whose firstname match the first word or lastname match the second word.
- `group_id`: get only users who are members of the given group

POST

Creates a user.

Parameters:

- `user` (required): a hash of the user attributes, including:
    - `login` (required): the user login
    - `password`: the user password
    - `firstname` (required)
    - `lastname` (required)
    - `mail` (required)
    - `auth_source_id`: authentication mode id

Example:

```
POST /users.xml
```

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<user>
  <login>jplang</login>
  <firstname>Jean-Philippe</firstname>
  <lastname>Lang</lastname>
  <password>secret</password>
  <mail>jp_lang@yahoo.fr</mail>
  <auth_source_id>2</auth_source_id>
</user>
```

JSON

```
{
    "user": {
        "login": "jplang",
        "firstname": "Jean-Philippe",
        "lastname": "Lang",
        "mail": "jp_lang@yahoo.fr",
        "password": "secret"
    }
}
```

> Response:

- `201 Created`: user was created
- `422 Unprocessable Entity`: user was not created due to validation failures (response body contains the error messages)

/users/:id.:format

GET

Returns the user details. You can use `/users/current.:format` for retrieving the user whose credentials are used to access the API.

> Parameters:

- `include` (optional): a coma separated list of associations to include in the response:
  - `memberships` : adds extra information about user's memberships and roles on the projects
  - `groups` (added in 2.1) : adds extra information about user's groups

> Examples:

```
GET /users/current.xml
```

Returns the details about the current user.

```
GET /users/3.xml?include=memberships,groups
```

Returns the details about user ID 3, and additional detail about the user's project memberships.

> Reponse:

```
<user>
  <id>3</id>
  <login>jplang</login>
  <firstname>Jean-Philippe</firstname>
  <lastname>Lang</lastname>
  <mail>jp_lang@yahoo.fr</mail>
  <created_on>2007-09-28T00:16:04+02:00</created_on>
  <last_login_on>2011-08-01T18:05:45+02:00</last_login_on>
  <custom_fields type="array" />
  <memberships type="array">
    <membership>
      <project name="Redmine" id="1"/>
      <roles type="array">
        <role name="Administrator" id="3"/>
        <role name="Contributor" id="4"/>
      </roles>
    </membership>
  </memberships>
  <groups type="array">
    <group id="20" name="Developers"/>
  </groups>
</user>
```

Depending on the status of the user who makes the request, you can get some more details:

- `api_key` : the API key of the user, visible for admins and for yourself (added in 2.3.0)
- `status` : a numeric id representing the status of the user, visible for admins only (added in 2.4.0). See app/models/principal.rb for a list of available statuses.

PUT

Updates a user.

> Example:

```
PUT /users/20.xml
```

> Parameters:

- `user` (required): a hash of the user attributes (same as for user creation)

DELETE

Deletes a user.

> Example:

```
DELETE /users/20.xml
```

> Response:

- `200 OK`: user was deleted

See also

- The Memberships API for adding or removing a user from a project.
- The Groups API for adding or removing a user from a group.

Time Entries

Listing time entries

```
GET /time_entries.xml
```

Returns time entries.

Showing a time entry

```
GET /time_entries/[id].xml
```

Returns the time entry of given id.

Creating a time entry

```
POST /time_entries.xml
```

Creates a time entry.

> Parameters:

- `time_entry` (required): a hash of the time entry attributes, including:
  - `issue_id` or `project_id` (only one is required): the issue id or project id to log time on
  - `spent_on`: the date the time was spent (default to the current date)
  - `hours` (required): the number of spent hours
  - `activity_id`: the id of the time activity. This parameter is required unless a default activity is defined in Redmine.
  - `comments`: short description for the entry (255 characters max)

> Response:

- `201 Created`: time entry was created
- `422 Unprocessable Entity`: time entry was not created due to validation failures (response body contains the error messages)

Updating a time entry

```
PUT /time_entries/[id].xml
```

Updates the time entry of given id.

> Parameters:

- `time_entry` (required): a hash of the time entry attributes (same as above)

> Response:

- `200 OK`: time entry was updated
- `422 Unprocessable Entity`: time entry was not updated due to validation failures (response body contains the error messages)

Deleting a time entry¶

```
DELETE /time_entries/[id].xml
```

Deletes the time entry of given id.

News

/news.:format

GET

Returns all news across all projects with pagination.

> Example:

```
GET /news.xml
```

/projects/:project_id/news.:format

GET

Returns all news from project with given id or identifier with pagination.

> Example:

```
GET /projects/foo/news.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<news type="array" limit="25" total_count="2" offset="0">
  <news>
    <id>54</id>
    <project name="Redmine" id="1"/>
    <author name="Jean-Philippe Lang" id="1"/>
    <title>Redmine 1.1.3 released</title>
    <summary/>
    <description>Redmine 1.1.3 has been released</description>
    <created_on>2011-04-29T14:00:25+02:00</created_on>
  </news>
  <news>
    <id>53</id>
    <project name="Redmine" id="1"/>
    <author name="Jean-Philippe Lang" id="1"/>
    <title>Redmine 1.1.2 bug/security fix released</title>
    <summary/>
    <description>Redmine 1.1.2 has been released</description>
    <created_on>2011-03-07T21:07:03+01:00</created_on>
  </news>
</news>
```

Issue Relations

/issues/:issue_id/relations.:format

GET

Returns the relations for the issue of given id (:issue_id).

> Example:

```
GET /issues/8470/relations.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<relations type="array">
  <relation>
    <id>1819</id>
    <issue_id>8470</issue_id>
    <issue_to_id>8469</issue_to_id>
    <relation_type>relates</relation_type>
    <delay/>
  </relation>
  <relation>
    <id>1820</id>
    <issue_id>8470</issue_id>
    <issue_to_id>8467</issue_to_id>
    <relation_type>relates</relation_type>
    <delay/>
  </relation>
</relations>
```

Note: when getting an issue, relations can also be retrieved in a single request using `/issues/:id.:format?include=relations`.

POST

Creates a relation for the issue of given id (:issue_id).

> Parameters:

- `relation` (required): a hash of the relation attributes, including:
  - `issue_to_id` (required): the id of the related issue
  - `relation_type` (required): the type of relation (in: "relates", "duplicates", "duplicated", "blocks", "blocked", "precedes", "follows")
  - `delay` (optional): the delay for a "precedes" or "follows" relation

> Response:

- `201 Created`: relation was created
- `422 Unprocessable Entity`: relation was not created due to validation failures (response body contains the error messages)

/relations/:id.:format

GET

Returns the relation of given id.

> Example:

```
GET /relations/1819.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<relation>
  <id>1819</id>
  <issue_id>8470</issue_id>
  <issue_to_id>8469</issue_to_id>
  <relation_type>relates</relation_type>
  <delay/>
</relation>
```

DELETE

Deletes the relation of given id.

> Response:

- `200 OK`: relation was deleted
- `422 Unprocessable Entity`: relation was not deleted (response body contains the error messages)

Versions

/projects/:project_id/versions.:format

GET

Returns the versions available for the project of given id or identifier (:project_id). The response may include shared versions from other projects.

Examples:

```
GET /projects/foo/versions.xml

GET /projects/1/versions.xml
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<versions type="array" total_count="34">
  <version>
    <id>1</id>
    <project name="Redmine" id="1"/>
    <name>0.7</name>
    <description/>
    <status>closed</status>
    <due_date>2008-04-28</due_date>
    <created_on>2008-03-09T12:52:06+01:00</created_on>
    <updated_on>2009-11-15T12:22:12+01:00</updated_on>
  </version>
  <version>
    <id>2</id>
    <project name="Redmine" id="1"/>
    <name>0.8</name>
    <description/>
    <status>closed</status>
    <due_date>2008-12-30</due_date>
    <created_on>2008-03-09T12:52:12+01:00</created_on>
    <updated_on>2009-11-15T12:22:12+01:00</updated_on>
  </version>
</versions>
```

POST

Creates a version for the project of given id or identifier (:project_id).

> Parameters:

- `version` (required): a hash of the version attributes, including:
  - `name` (required)
  - `status`: the status of the version in: `open (default), locked, closed`
  - `sharing`: the version sharing in: `none (default), descendants, hierarchy, tree, system`
  - `due_date`
  - `description`

> Response:

- `201 Created`: version was created
- `422 Unprocessable Entity`: version was not created due to validation failures (response body contains the error messages)

/versions/:id.:format

GET

Returns the version of given id.

> Example:

```
GET /versions/2.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<version>
  <id>2</id>
  <project name="Redmine" id="1"/>
  <name>0.8</name>
  <description/>
  <status>closed</status>
  <due_date>2008-12-30</due_date>
  <created_on>2008-03-09T12:52:12+01:00</created_on>
  <updated_on>2009-11-15T12:22:12+01:00</updated_on>
</version>
```

PUT

Updates the version of given id

> Parameters:

Same as version creation

> Response:

- `200 OK`: version was updated
- `422 Unprocessable Entity`: version was not updated due to validation failures (response body contains the error messages)

DELETE

Deletes the version of given id.

> Response:

- `200 OK`: version was deleted
- `422 Unprocessable Entity`: version was not deleted (response body contains the error messages)

Wiki Pages

Getting the pages list of a wiki

```
GET /projects/foo/wiki/index.xml
```

Returns the list of all pages in a project wiki.

> Response:

```
<?xml version="1.0"?>
<wiki_pages type="array">
  <wiki_page>
    <title>UsersGuide</title>
    <version>2</version>
    <created_on>2008-03-09T12:07:08Z</created_on>
    <updated_on>2008-03-09T23:41:33+01:00</updated_on>
  </wiki_page>
  ...
</wiki_pages>
```

Getting a wiki page

```
GET /projects/foo/wiki/UsersGuide.xml
```

Returns the details of a wiki page.

> Includable:

- attachments

> Response:

```
<?xml version="1.0"?>
<wiki_page>
  <title>UsersGuide</title>
  <parent title="Installation_Guide"/>
  <text>h1. Users Guide
  ...
  ...</text>
  <version>22</version>
  <author id="11" name="John Smith"/>
  <comments>Typo</comments>
  <created_on>2009-05-18T20:11:52Z</created_on>
  <updated_on>2012-10-02T11:38:18Z</updated_on>
</wiki_page>
```

Getting an old version of a wiki page

```
GET /projects/foo/wiki/UsersGuide/23.xml
```

Returns the details of an old version of a wiki page.

> Includable:

- attachments

> Response:

Same as above.

Creating or updating a wiki page

```
PUT /projects/foo/wiki/UsersGuide.xml
<?xml version="1.0"?>
<wiki_page>
  <text>Example</text>
  <comments>Typo</comments>
</wiki_page>
```

Creates or updates a wiki page.

When updating an existing page, you can include a `version` attribute to make sure that the page is a specific version when you try to update it (eg. you don't want to overwrite an update that would have been done after you retrieved the page). Example:

```
PUT /projects/foo/wiki/UsersGuide.xml
```

```
<?xml version="1.0"?>
<wiki_page>
  <text>Example</text>
  <comments>Typo</comments>
  <version>18</version>
</wiki_page>
```

This would update the page if its current version is 18, otherwise a `409 Conflict` error is returned.

> Response:

- `200 OK`: page was updated
- `201 Created`: page was created
- `409 Conflict`: occurs when trying to update a stale page (see above)
- `422 Unprocessable Entity`: page was not saved due to validation failures (response body contains the error messages)

Deleting a wiki page

```
DELETE /projects/foo/wiki/UsersGuide.xml
```

Deletes a wiki page, its attachments and its history. If the deleted page is a parent page, its child pages are not deleted but changed as root pages.

> Response:

- `200 OK`: page was deleted

Queries

/queries.:format

GET

Returns the list of all custom queries visible by the user (public and private queries) for all projects.

> Examples:

```
GET /queries.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<queries type="array" total_count="5" limit="25" offset="0">
  <query>
    <id>84</id>
    <name>Documentation issues</name>
    <is_public>true</is_public>
    <project_id>1</project_id>
  </query>
  <query>
    <id>1</id>
    <name>Open defects</name>
    <is_public>true</is_public>
    <project_id>1</project_id>
  </query>
```

```
</queries>
```

Armed with a query id, you can get the corresponding issue list using:

```
GET /issues.xml?query_id=:id

GET /issues.xml?query_id=:id&project_id=foo
```

Attachments

To attach files though the API, please see Attaching files in general topics.

/attachments/:id.:format

GET

Returns the description of the attachment of given id.
The file can actually be downloaded at the URL given by the `content_url` attribute in the response.

> Example:

```
GET /attachments/13.xml
```

> Response:

```
<attachment>
  <id>6243</id>
  <filename>test.txt</filename>
  <filesize>124</filesize>
  <content_type>text/plain</content_type>
  <description>This is an attachment</description>

<content_url>http://localhost:3000/attachments/download/6243/test.txt</conten
t_url>
  <author name="Jean-Philippe Lang" id="1"/>
  <created_on>2011-07-18T22:58:40+02:00</created_on>
</attachment>
```

Note: when getting an issue through the API, its attachments can also be retrieved in a single request using `GET /issues/:id.:format?include=attachments`.

Issue Statuses

/issue_statuses.:format¶

GET

Returns the list of all issue statuses.

> Examples:

```
GET /issue_statuses.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<issue_statuses type="array">
  <issue_status>
    <id>1</id>
    <name>New</name>
    <is_default>true</is_default>
    <is_closed>false</is_closed>
  </issue_status>
  <issue_status>
    <id>2</id>
    <name>Closed</name>
    <is_default>false</is_default>
    <is_closed>true</is_closed>
  </issue_status>
</issue_statuses>
```

Rest Trackers

/trackers.:format¶

GET

Returns the list of all trackers.

> Examples:

```
GET /trackers.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<trackers type="array">
  <tracker>
    <id>1</id>
    <name>Defect</name>
  </tracker>
  <tracker>
    <id>2</id>
    <name>Feature</name>
  </tracker>
</trackers>
```

Enumerations

/enumerations/issue_priorities.:format

GET

Returns the list of issue priorities.

> Examples:

```
GET /enumerations/issue_priorities.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<issue_priorities type="array">
  <issue_priority>
```

```
    <id>3</id>
    <name>Low</name>
    <is_default>false</is_default>
  </issue_priority>
  <issue_priority>
    <id>4</id>
    <name>Normal</name>
    <is_default>true</is_default>
  </issue_priority>
  ...
</issue_priorities>
```

/enumerations/time_entry_activities.:format

GET

Returns the list of time entry activities.

> Examples:

```
GET /enumerations/time_entry_activities.xml
```

> Response:

```
<time_entry_activities type="array">
  <time_entry_activity>
    <id>8</id>
    <name>Design</name>
    <is_default>false</is_default>
  </time_entry_activity>
  ...
</time_entry_activities>
```

Issue Categories

- PUT
- DELETE

/projects/:project_id/issue_categories.:format

GET

Returns the issue categories available for the project of given id or identifier (:project_id).

Examples:

```
GET /projects/foo/issue_categories.xml

GET /projects/1/issue_categories.xml
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<issue_categories type="array" total_count="2">
  <issue_category>
    <id>57</id>
    <project name="Foo" id="17"/>
    <name>UI</name>
    <assigned_to name="John Smith" id="22"/>
  </issue_category>
  <issue_category>
    <id>58</id>
    <project name="Foo" id="17"/>
    <name>Test</name>
  </issue_category>
</issue_categories>
```

POST

Creates an issue category for the project of given id or identifier (:project_id).

Parameters:

- `issue_category` (required): a hash of the issue category attributes, including:
  - name (required)
  - `assigned_to_id`: the id of the user assigned to the category (new issues with this category are assigned by default to this user)

Response:

- `201 Created`: issue category was created

- `422 Unprocessable Entity`: issue category was not created due to validation failures (response body contains the error messages)

/issue_categories/:id.:format

GET

Returns the issue category of given id.

> Example:

```
GET /issue_categories/2.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<issue_category>
  <id>2</id>
  <project name="Redmine" id="1"/>
  <name>UI</name>
</version>
```

PUT

Updates the issue category of given id

> Parameters:

Same as issue category creation

> Response:

- `200 OK`: issue category was updated
- `422 Unprocessable Entity`: issue category was not updated due to validation failures (response body contains the error messages)

DELETE

Deletes the issue category of given id.

> Parameters:

- `reassign_to_id` (optional): when there are issues assigned to the category you are deleting, this parameter lets you reassign these issues to the category with this id

> Example:

```
DELETE /issue_categories/2.xml
DELETE /issue_categories/2.xml?reassign_to_id=1
```

> Response:

- `200 OK`: issue category was deleted

Roles

/roles.:format

GET

Returns the list of roles.

> Examples:

```
GET /roles.xml
```

> Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<roles type="array">
  <role>
    <id>1</id>
    <name>Manager</name>
  </role>
  <role>
    <id>2</id>
    <name>Developer</name>
  </role>
</roles>
```

/roles/[id].:format

GET

Returns the list of permissions for a given role (2.2.0).

> Examples:

```
GET /roles/5.xml
```

> Response:

```
<role>
  <id>5</id>
  <name>Reporter</name>
  <permissions type="array">
    <permission>view_issues</permission>
    <permission>add_issues</permission>
    <permission>add_issue_notes</permission>
    ...
  </permissions>
</role>
```

Groups

/groups.:format

GET

Returns the list of groups.

> Example:

```
GET /groups.xml
```

Response:

```
<groups type="array">
  <group>
    <id>53</id>
    <name>Managers</name>
  </group>
  <group>
    <id>55</id>
    <name>Developers</name>
  </group>
</groups>
```

POST

Creates a group.

Parameters:

- `group` (required): a hash of the group attributes, including:
  - name (required): the group name
  - `user_ids`: ids of the group users (an empty group is created if not provided)

Example:

```
POST /groups.xml

<group>
  <name>Developers</name>
  <user_ids>
    <user_id>3</user_id>
    <user_id>5</user_id>
  </user_ids>
</group>
POST /groups.json

{
  "group": {
    "name": "Developers",
    "user_ids": [ 3, 5 ]
  }
}
```

> Response:

- `201 Created`: group was created
- `422 Unprocessable Entity`: group was not created due to validation failures (response body contains the error messages)

/groups/:id.:format

GET

Returns details of a group.

> Parameters:

- `include` (optional): a coma separated list of associations to include in the response:
    - `users`
    - `memberships`

> Example:

```
GET /groups/20.xml?include=users
```

> Response:

```
<group>
  <id>20</id>
  <name>Developers</name>
  <users type="array">
    <user id="5" name="John Smith"/>
    <user id="8" name="Dave Loper"/>
  </users>
</group>
```

PUT

Updates an existing group.

DELETE

Deletes an existing group.

/groups/:id/users.:format

POST

Adds an existing user to a group.

> Parameters:

- `user_id` (required): id of the user to add to the group.

> Example:

```
POST /groups/10/users.xml


<user_id>5</user_id>
```

> Response:

- `200 OK`: user was added to the group

/groups/:id/users/:user_id.:format

DELETE¶

Removes a user from a group.

> Example:

```
DELETE /groups/10/users/5.xml
```

> Response:

- `200 OK`: user was removed to the group

Custom Fields

/custom_fields.:format

GET

Returns all the custom fields definitions.

> Examples:

```
GET /custom_fields.xml
```

Response:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<custom_fields type="array">
  <custom_field>
    <id>1</id>
    <name>Affected version</name>
    <customized_type>issue</customized_type>
    <field_format>list</field_format>
    <regexp/>
    <min_length/>
    <max_length/>
    <is_required>true</is_required>
    <is_filter>true</is_filter>
    <searchable>true</searchable>
    <multiple>true</multiple>
    <default_value/>
    <visible>false</visible>
    <possible_values type="array">
      <possible_value>
        <value>0.5.x</value>
      </possible_value>
      <possible_value>
        <value>0.6.x</value>
      </possible_value>
  <custom_field>
  <custom_field>
    ...
  </custom_field>
</custom_fields>
```

The `customized_type` attribute indicates which type of object the custom field applies to (eg. issue, project, time_entry...).