

## Introduction

Daisy contains a HTTP+XML interface, which is an interface to talk to the repository server by exchanging XML messages over the HTTP protocol. This interface offers full access to all functionality of the repository.

The HTTP protocol is a protocol that allows to perform a limited number of methods (Daisy uses GET, POST and DELETE) on an unlimited number of resources, which are identified by URIs. The GET method is used to retrieve a representation of the addressed resource, POST to trigger a process that modifies the addressed resource, and DELETE to delete a resource.

With HTTP, all calls are independent of each other, there is no session with the server.

The Daisy HTTP interface listens by default on port 9263. You can easily try it out, for example if Daisy is running on your localhost, just enter the URL below in the location bar of the browser, and press enter. The browser will then send a GET request to the server. The example given here is a request to execute a query (written in the Daisy Query Language). This request doesn't require an XML payload, all parameters are specified as part of the URL. Note that spaces in an URL must be encoded with a plus symbol.

```
http://localhost:9263/repository/query?q=select+id,name+where+true&locale=en
```

The browser will ask a user name and password, enter your Daisy repository username and password (e.g., the one you otherwise use to log in on the Daisy Wiki), or use the user name "guest" and password "guest" (only works if you installed the Daisy Wiki). The browser will show the XML response received from the server (in some browsers, you might need to do "view source" to see it).

Not all operations can be performed as easily as the above example: some require POST or DELETE as method, some require an XML document in the body of the request, and some even require a multipart-formatted request body (the document create and update operations, which need to upload the binary part data next to the XML message). If you have a programming language with a decent HTTP client library, none of this should be a problem.

## Authentication

All requests require authentication. Authentication is done using BASIC authentication.

If you want to log in as another role then the default role of a user, append "@<roleid>" to the login (without the quotes). Note that it must be the id of the role, not its name. For example, if your default role is not Administrator (ID: 1), but you would like to perform the request as Administrator, and your login is "jules", you would use "jules@1". When the login itself contains an @-symbol, it must be escaped by doubling it (i.e. each @ should be replaced with @@). Multiple active roles can be specified using a comma-separated list, e.g. "jules@1,105".

## Robustness

The current implementation doesn't do (many) checks on the XMLs posted as part of a HTTP request. This means that for example missing elements or attributes might simply cause little-descriptive (but harmless) "NullPointerExceptions" to occur.

The reason for this is that we use the HTTP API mostly via the repository Java client, which generates valid messages for us.

Since the XML posted to a resource is usually the same as the XML retrieved via GET on the same resource, it is easy to get examples of correct XML messages. XML Schemas are also available (see further on), though being schema-valid doesn't necessarily imply the message is correct.

## Error handling

If a response was handled correctly, the server will answer with HTTP status code 200 (OK). If the status code has another value, it means something went wrong.

For errors generated explicitly, or when a Java exception occurs, an XML message is created describing the exception, and is returned with a status code 202 (Accepted). The XML message consists of an <error> root element, with as child either a <description> element or a <cause> element. The <description> element contains a simple string describing the error. The <cause> element is used in case a Java exception was handled, and contains further elements describing the exception (including stacktrace), and can include <cause> elements recursively describing the "causing" exceptions of that exception. To see an example of this, simply do a request for a non-existing resource, e.g.:

```
http://localhost:9263/repository/document/99999999
```

(assuming there is no document with ID 99999999)

When executing a method (GET, POST, DELETE, ...) on a resource that doesn't support that method you will get status code 405 (Method Not Allowed).

Incorrect or missing authentication information will give status code 401 (Unauthorized).

Missing request parameters, or invalid ones (eg. giving a string where a number was expected) will give status code 400 (Bad Request).

Doing a request for a non-existing resource will give status code 404 (Not Found)

## Intro to the reference

The rest of this document describes the available URLs, the operations that can be performed upon them, and the format of the XML messages. The descriptions can be dense, the current goal of this document is just to give a broad overview, more details might be added later. You can always ask for more information on the Daisy Mailing List.

You can also investigate how things are supposed to work by monitoring the HTTP traffic between the Daisy Wiki and the Daisy Repository Server.

Sometimes XML Schema files are referenced, these can be found in the Daisy source distribution.

## Core Repository Interface

### Documents

On many document-related resources, request parameters called branch and language can be added (this will be mentioned in each case). The value of these parameters can be either a name or ID of a branch or language. If not specified, the branch "main" and the language "default" are assumed.

#### /repository/document

This resource represents the set of all documents. GET is not supported on this resource (you can retrieve a list of all documents using a query).

POST on this resource is used to create a new document, which also implies the creation of a document variant, since a document cannot exist without a document variant. The payload should be a multipart request having one multipartrequest-part (we use this long name to distinguish with Daisy's document parts) containing the XML description of the new document, and other multipartrequest-parts containing the content of the document parts (if any). The multipartrequest-part containing the XML should be called "xml", and should conform to the document.xsd schema. The part elements in the XML should have dataRef attributes whose value is the name of the multipartrequest-part containing the data for that part.

The server will return the XML description of the newly created document as result. This XML will, among other things, have the id attribute completed with the ID of the new document.

## Example scenario: creating a new document

This example illustrates how to create a new document in the repository over the HTTP interface using the [curl](#)<sup>1</sup> tool. Curl is a handy command-line tool to do HTTP (and other) requests, and is standard available on many Linux distributions (it exists for Windows too).

Suppose we want to create a new document of type 'SimpleDocument' (as used in the Daisy Wiki), with the part 'SimpleDocumentContent'. We start by creating the XML description of the document, and save it in a file called newdoc.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:document
  xmlns:ns="http://outerx.org/daisy/1.0"
  name="My test doc"
  typeId="2"
  owner="3"
  validateOnSave="true"
  newVersionState="publish"
  retired="false"
  private="false"
  branchId="1"
  languageId="1"
  >
  <ns:customFields/>
  <ns:collectionIds>
    <ns:collectionId>1</ns:collectionId>
  </ns:collectionIds>
  <ns:fields/>
  <ns:parts>
    <ns:part mimeType="text/xml" typeId="2" dataRef="data1"/>
  </ns:parts>
  <ns:links/>
</ns:document>
```

Some items in the above XML will need to be changed for your installation:

- document/@typeId: this is the ID of the document type to use for the document. In my installation the ID for 'SimpleDocument' is 2, but you need to check that on your installation. You can see this either on the administration pages of the Daisy Wiki, or by doing a request to <http://localhost:9263/repository/schema/documentType>
- document/@owner: this is the ID of the owner of the document, this should be an existing user ID. You can again see the user IDs on the administration pages of the Daisy Wiki.
- document/parts/part/@typeId: this is the ID of the part type. In my installation the ID for 'SimpleDocumentContent' is 2, but you need to check that on your installation. You can see this either on the administration pages of the Daisy Wiki, or by doing a request to <http://localhost:9263/repository/schema/partType>

Now we need to create a file containing the content of the part we're going to add. For example create a file called 'mynewfile.xml' and put the following in it:

```
<html>
  <body>
```

```
<h1>Hi there!</h1>

<p>This is a test document.</p>

</body>
</html>
```

Finally we are ready to create the document using curl:

```
curl --basic
      --user testuser:testuser
      --form xml=@newdoc.xml
      --form data1=@mynewfile.xml
      http://localhost:9263/repository/document
```

You need to enter all arguments on one line of course, and change user, password and server URLs as appropriate for your installation. Note that the form parameter 'data1' corresponds to the dataRef attribute in the newdoc.xml file (you can choose any name you want for these, if you have multiple parts use different names)

`/repository/document/<id>`

`<id>` should be replaced with the ID of an existing document.

### Retrieving a document

GET on this resource retrieves an XML description of a document, with a certain variant of the document. The XML will contain the data of the most recent version of the document variant. The (binary) part data is not embedded in the XML, but must be retrieved separately using the following URL (described further on):

```
/repository/document/<document-id>/version/<version-id>/part/<parttype-id>/data
```

To specify the document variant, add the optional request parameters `branch` and `language`.

### Creating a document or adding a document variant

POST on this resource is used to update a document (and/or document variant), or to add a new variant to it. When adding a new variant there are two possibilities: initialise the new variant with the content of an existing variant, or create a new variant from scratch. We now describe these three distinct cases.

To update an existing document (document variant), the format of the POST is similar as when creating a document, that is, it should contain a multipart-format body. The XML in this case should be an updated copy of the XML retrieved via the GET on this resource. Unmodified parts don't need to be uploaded again.

To create a new variant from scratch, again the POST data is similar as when creating a new document. In addition, three request parameters must be specified:

- `createVariant` with the value `yes`
- `startBranch`
- `startLanguage`

Although the variant is created from scratch, it is only possible to add a new variant to a document if you have at least read access to an existing variant. The new variant to be created is specified by the `branchId` and `languageId` attributes within the posted XML.

Creating a new variant based on an existing variant is rather different. In this case no XML body or multipart-request must be done, but a POST operation with the following request parameters:

- `startBranch`
- `startLanguage`
- `startVersion`: specify `-1` for last version, `-2` for live version
- `newBranch`

- newLanguage

These parameter names explain themselves I think. The branches and languages can be specified either by name or ID.

### Deleting a document or a document variant

DELETE on this resource permanently deletes the document. This will delete the document and all its variants.

To delete only one variant of the document, specify the request parameters branch and language.

### /repository/document/<id>/version

GET on this resource returns a list of all versions in a document variant as XML. For each version, only some basic information is included (the things typically needed to show a version overview page).

To specify the document variant, add the optional request parameters branch and language.

### /repository/document/<id>/version/<id>

GET on this resource returns the full XML description of this version. As when requesting a document, the actual binary part data is not embedded in the XML but has to be retrieved separately.

POST on this resource is used to modify the version state (which is the only thing of a version that can be modified, other than that, versions are read-only once created). The request should have two parameters:

- action=changeState
- newState=publish|draft

For both the GET and POST methods, add the optional request parameters branch and language to specify the variant.

### /repository/document/<id>/version/<id>/part/<id>/data

GET on this resource retrieves the data of a part in a certain version of a document. The meaning of the <id>'s is as follows:

1. The first <id> is the document ID
2. The second <id> the version ID (1, 2, 3, ...) or the string "last" to signify the last version
3. The third <id> is the part type ID or the part type name of the part to be retrieved (if the first character is a digit, it is supposed to be an ID. Part type names cannot begin with a digit).

To specify the document variant, add the optional request parameters branch and language.

See also the /publisher/blob resource, which sets correct mime type, last modified and content length headers.

### /repository/document/<id>/lock

See the lock.xsd file for the XML Schema of the XML used to interact with this resource.

GET on this resource returns information about the lock, if any.

POST on this resource is used to create a lock. In this case, all attributes in the XML must have a value except for the hasLock attribute.

DELETE on this resource is used to remove the lock (if any). No request body is required.

For all three methods, the returned result is the XML description of the lock after the performed operation (possibly describing that there is no lock).

A lock applies to a certain variant of a document. To specify the document variant, add the optional request parameters branch and language.

## /repository/document/<id>/comment

See comment.xsd for the XML Schema of the messages.

GET on this resource returns the list of comments for a document variant. To specify the document variant, add the optional request parameters branch and language.

POST on this resource creates a new comment. The branch and language are in this case specified in the XML message.

## /repository/document/<id>/comment/<id>

DELETE on this resource deletes a comment. The second <id> is the ID of the comment. To specify the document variant, add the optional request parameters branch and language.

Other methods are not supported on this resource.

## /repository/document/<id>/availableVariants

A GET on this resource returns the list of variants that exist for this document.

## Schema Management

### /repository/schema/(part|field|document)Type

These resources represent the set of part, field and document types.

POST to these resources is used to create a new part, field or document type. The request body should contain an XML message conforming to the schemas found in fieldtype.xsd, parttype.xsd or documenttype.xsd.

### Example scenario

This example illustrates how to create a new field type (with a selection list), simply by using the well-known "wget" tool.

This is the XML that we'll send to the server:

```
<?xml version="1.0"?>
<fieldType name="myNewField" valueType="string" deprecated="false"
  aclAllowed="false" size="0"
  xmlns="http://outerx.org/daisy/1.0">
  <labels>
    <label locale="">My New Field</label>
  </labels>
  <descriptions>
    <description locale="">This is a test field</description>
  </descriptions>
  <selectionList>
    <listItem>
      <labels/>
      <string>value 1</string>
    </listItem>
    <listItem>
      <labels/>
      <string>value 2</string>
    </listItem>
  </selectionList>
</fieldType>
```

Let's say we save this in a file called newfieldtype.xml. We can then create the field by executing:

```
wget --post-file=newfieldtype.xml
--http-user=testuser@1
--http-passwd=testuser
```

```
http://localhost:9263/repository/schema/fieldType
```

This supposes that "testuser" exists and has the Administrator role, which is required for creating field types.

Wget will save the response from the server in a file called "fieldType". The response is the same XML but now with some additional attributes such as the assigned ID. The response XML isn't pretty formatted, if you have libxml installed you can view it pretty using:

```
xmllint --format fieldType
```

## /repository/schema/(part|field|document)Type/<id>

GET on these resources retrieves the XML representation of a part, field or document type.

POST on these resources updates a part, field or document type. The request body should then contain an altered variant of the XML retrieved via GET.

DELETE on these resources deletes them. Note that deleting types is only possible if they are not in use any more by any version of any document.

### Example scenario

Let's take the previous field type example again, and add an additional value to the selection list. We first retrieve the XML for the field type (check the XML response of the previous sample to know the ID of the created field type):

```
wget http://localhost:9263/repository/schema/fieldType/1
```

This will save a file called "1" (if that was the requested ID). To make it easier to work with, do a:

```
xmllint --format 1 > updatedfieldtype.xml
```

and change the updatedfieldtype.xml with an additional value in the selection list:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:fieldType xmlns:ns="http://outerx.org/daisy/1.0" size="0" updateCount="1"
  aclAllowed="false" deprecated="false" valueType="string"
  name="myNewField" lastModifier="3"
  lastModified="2004-09-09T09:06:51.032+02:00" id="1">
  <ns:labels>
    <ns:label locale="">My New Field</ns:label>
  </ns:labels>
  <ns:descriptions>
    <ns:description locale="">This is a test field</ns:description>
  </ns:descriptions>
  <ns:selectionList>
    <ns:listItem>
      <ns:labels/>
      <ns:string>value 1</ns:string>
    </ns:listItem>
    <ns:listItem>
      <ns:labels/>
      <ns:string>value 2</ns:string>
    </ns:listItem>
    <ns:listItem>
      <ns:labels/>
      <ns:string>value 3</ns:string>
    </ns:listItem>
  </ns:selectionList>
</ns:fieldType>
```

And then do:

```
wget --post-file=updatedfieldtype.xml
  --http-user=testuser@1
  --http-passwd=testuser
```

/repository/schema/(part|field|document)TypeByName/<name>

GET on this resource retrieves a part, field or document type by its name.

You cannot POST on this resource, to update the type, use the previous (ID-based) resource.

## Access Control Management

/repository/acl/<id>

/repository/filterDocumentTypes

## Collection Management

/repository/collection/<id>

/repository/collectionByName/<name>

/repository/collection

## User Management

/repository/user

/repository/role

/repository/user/<id>

/repository/role/<id>

/repository/userByLogin/<login>

/repository/roleByName/<name>

/repository/usersByEmail/<email>

/repository/userIds

## Variant Management

/repository/branch

/repository/branch/<id>

/repository/branchByName/<name>

/repository/language

/repository/language/<id>

/repository/languageByName/<name>

## Querying

---

**Repository/query**

ID: 21-daisy | Version: 1 | Date: 7/15/10 9:26:46 AM

GET on this resource is used to perform queries using the [Daisy Query Language](#)<sup>2</sup>.

Required parameters:

- q=<some daisy query>
- locale=en-US : the locale to be used to format the result data (influences date and number formatting)

Optional parameters:

- returnKeys=true|false : false by default, if true will return only the keys of the document variants satisfying the query (that is: the tripple document ID, branch, language), instead of full results. (Remember that the result of a query is a set of document variants).
- extraCondition=<conditional expression> : will combine this conditional expression with the existing where conditions of the given query, using AND. For example, this is used in the Daisy Wiki to limit query results to a certain collection. The extraCondition is then something like: InCollection('mycollection'). Note that this is different from simply appending "AND" and the conditional expression to the end of the query, since the query can include order by and other clauses at the end.

## /repository/facetedQuery

Used to perform a query for which the result contains the distinct values for the different items returned by the query. This allows to build a "faceted navigation" front end.

The query parameters are specified in an XML document which should be posted to this resource. The format of the XML is defined in facetedquery.xsd.

## Other

### /repository/userinfo

GET on this resource returns some information about the authenticated user. Takes no parameters.

### /repository/comments

Usually comments are retrieved via the document they belong to, but it is also possible to get all comments of a user by doing a GET on this resource. Parameters:

- **visibility** (optional): one of: public, private, editors. Causes only comments with the given visibility to be included.

## Navigation Manager Extension

See also [navigation](#)<sup>3</sup>.

### /navigation

GET on this resource retrieves a navigation tree (customised for the authenticated user).

Parameters, all required unless indicated otherwise:

- navigationDocId
- navigationDocBranch: branch ID or name
- navigationDocLanguage: language ID or name
- All or none of the following:
  - activeDocumentId: ID of the document to be selected
  - activeDocumentBranch: branch ID or name
  - activeDocumentLanguage: language ID or name

- `activePath`: suggested path in the navigation tree where to look for the `activeDocument` (the same document might appear in multiple locations in the tree) (not required)
- `contextualized=true|false`: if true, only nodes leading to the active document will be opened, others will be closed (thus, their children will not be included in the navigation tree output)
- `handleErrors=true|false`: if false, when an exception occurs, the result status code from the server will not be "200 OK" but "202 Accepted" with the body containing an XML representation of the Java exception. If true and exception occurs, a normal 200 response will be given and the body will contain `<n:navigationTree><n:navigationTreeError/></n:navigationTree>`, where the "n" prefix maps to the navigation tree result namespace. The `navigationTreeError` element is currently empty, but might in the future contain a description of the error. The `handleErrors=true` parameter is useful to avoid that the rendering of a page fails completely when generating the navigation tree fails.

## /navigationPreview

This resource allows to generate a navigation tree from a navigation source description specified as part of the request. This is used in the Daisy Wiki application to try out navigation trees before saving them.

Parameters:

- `navigationXml`: the navigation tree input XML
- `branch`
- `language`

## /navigationLookup

Resolves a path against the navigation tree and returns the result of that lookup as an XML message. For more details, see the Java API (e.g. the class `NavigationLookupResult`).

Required parameters:

- `navigationDocId`
- `navigationDocBranch`
- `navigationDocLanguage`
- `navigationPath`

## Publisher Extension

The purpose of the Publisher Extension component is to return in one call all the data needed to publish pages in the Daisy Wiki (or other front end applications).

### /publisher/request

To this resource you can POST a publisher request. A publisher request takes the form of an XML document, and is described in detail [over here](#)<sup>4</sup>.

### /publisher/blob

GET on this resource retrieves a blob (the data of a part).

Required parameters:

- `documentId`
- `branch`
- `language`
- `version`: a version ID, or "last" or "live"
- `partType`: ID or name

The last modified, content type and content length headers are set.

## Email Notifier Extension

The Email Notifier extension component makes available resources for managing the email subscriptions.

`/emailnotifier/subscription/<id>`

`<id>` is the ID of a user.

GET, POST, DELETE supported. XML Schema see `subscription.xsd`

`/emailnotifier/subscription`

GET on this resource returns all the subscriptions.

`/emailnotifier/(document|schema|user|acl|collection)EventsSubscribers`

GET on this resource returns all subscribers for the kind of event as specified in the request path. The returned information for each subscriber includes the user ID and the locale for the subscription.

In the case of `documentEventSubscribers`, the following additional request parameters are required:

- `documentId`
- `branch`
- `language`
- `collectionIds`: comma separated list of IDs of collections the document belongs to.

The returned subscribers are then those that are explicitly subscribed for changes to that document, or those who are subscribed to a collection to which the document belongs, or those that are subscribed to all collections.

`/emailnotifier/documentSubscription/<documentId>`

This resource allows to manage document-based subscriptions without having to go through the full subscriptions.

Using POST on this resource allows to add or remove a subscription for the specified document for some user. Required parameters:

- `action`: add or delete
- `userId`
- `branch` (an ID)
- `language` (an ID)

Using DELETE on this resource removes the subscriptions for the specified document for all users. This is useful to cleanup subscriptions when a document gets deleted. If the `branch` and `language` parameters are missing, the subscriptions for all variants fo the document will be removed, otherwise only for the specified variant.

`/emailnotifier/documentSubscription/<documentId>/<userId>`

This resource allows to quickly check if a user is subscribed for notifications to a certain document (using GET). Request parameters `branch` and `language` are required, specifying the ID of the branch and language.

`/emailnotifier/collectionSubscription/<collectionId>`

Only DELETE is supported on this resource, and deletes all subscriptions for the specified collection for all users.

## Mailer Extension

The mailer extension allows to send emails. Only Administrators can do this.

### /emailer

A POST to this resource will send an email, the following parameters are required:

- to
- subject
- messageText

## Document Task Manager Extension

See also [Document Task Manager](#)<sup>5</sup>.

### /doctaskrunner/task

A GET on this resource retrieves all existing tasks for the current user, or the tasks of all users if the role is Administrator.

A POST on this resource is used to create a new task, in which case the body must contain an XML document describing the task (see also taskdescription.xsd).

### /doctaskrunner/task/<id>

A GET on this resource retrieves information about a task.

A POST on this resource in combination with a request parameter "action" with value "interrupt" interrupts a task.

A DELETE on this resource deletes the persistent information about this task.

### /doctaskrunner/task/<id>/docdetails

A GET on this resource retrieves detailed information about the execution of the task on the documents.

1. <http://curl.haxx.se/>
2. [/daisy-docs-1\\_3/repository/general/9-daisy.html](/daisy-docs-1_3/repository/general/9-daisy.html)
3. [/daisy-docs-1\\_3/daisywiki/general/12-daisy.html](/daisy-docs-1_3/daisywiki/general/12-daisy.html)
4. [/daisy-docs-1\\_3/repository/general/194-daisy.html](/daisy-docs-1_3/repository/general/194-daisy.html)
5. [/daisy-docs-1\\_3/repository/general/157-daisy.html](/daisy-docs-1_3/repository/general/157-daisy.html)