

---

# Dove 1.1 XML API

Pierre van Rooden

MMBase 1.5 includes a beta version of Dove. Version 1.0 was released with MMBase 1.5.1. Version 1.1 is included with MMBase 1.6.

This software is OSI Certified Open Source Software. OSI Certified is a certification mark of the Open Source Initiative.

The license (Mozilla version 1.0) can be read at the MMBase site. See <http://www.mmbase.org/license>

## Table of Contents

Introduction .....	1
Passing Commands .....	2
Security .....	3
Errors .....	3
Commands .....	4
getdata .....	5
getrelations .....	6
Nesting calls (getdata/getrelations) .....	8
getnew .....	10
getnewrelation .....	11
getconstraints .....	12
getlist .....	14
put .....	16

## Introduction

The Dove is a support class for MMBase, which adds a protocol for communication with, and passing commands to, the MMBase system using an XML format. The current implementation (`org.mmbase.applications.dove.Dove`) makes use of the MMCI to communicate with MMBase. Alternate implementations (i.e. versions that work on the core) can be implemented by extending the `AbstractDove` class. Every implementation should at the least support the XML format presented here, though it is possible that implementations add custom commands or functionality. Commands are run by passing an XML DOM root element to an instance of Dove (the request). The calling member function returns an `Element` that contains the result of the passed command (the response). This document focuses on the structure of the request and response XML (the Dove Communication API). The actual mechanics for instantiation of the Dove class and calling its member functions are documented in the Dove class API.

The version noted here is version 1.1, which is included with MMBase version 1.6 and higher.

### A Note On XML Validation

The XML you pass to the Dove is an already parsed tree, to support varying methods of obtaining the XML. This means that XML validation needs to be done by the calling code.

The current Dove implementation does not assume that the passed DOM tree is validated against the required DTD. Unsupported attributes are ignored, but the code does perform some syntax checks and returns error tags when it does not recognize a tag. Future implementations may forego this kind of 'validation', or may fail if it does not recognize attributes, so assuming this behavior is discouraged, and is not a part of the API specification.

## Note

Specific behavior of the default Dove implementation are included in this document as a note (like this).

### A Note on Syntax

The commands handled below will include a syntax description and, where appropriate, an example.

The syntax descriptions will be in a DTD format, and shown as source text (like

Only syntax relevant to the appropriate section will be shown (the actual DTD differs slightly as some elements are used in various places).

The examples are fictitious XML documents. To shorten the examples, the XML declaration and DOCTYPE elements are not included.

The XML examples, or fractions thereof, are shown as source text (like this).

This document deals with three versions of Dove: 1.0 beta (included in MMBase 1.5), 1.0 (included in MMBase 1.5.1), and 1.1. (MMBase 1.6).

Where appropriate, notes will be used to indicate which versions support certain functionality.

## Passing Commands

The root element of the request xml is the `request` tag.

Each request tag contains at most one `security` tag (containing user login information), and one or more dove command tags (each tag being one command). If more than one command is passed, the commands are handled in the order that they appear in the xml.

```
<!ELEMENT request (security?,(getdata|getnew|getnewrelation|getrelations| getco
```

The commands that should be comprehended are : `getdata`, `getnew`, `getnewrelation`, `getrelations`, `getconstraints`, `getlist`, and `put`.

## Note

Custom Dove implementations may implement additional commands.

The command tags support an `id` attribute. The attribute has no significance to the Dove, but is meant to support a client-generated id system. When an `id` attribute is provided, this attribute is also returned in the appropriate tag in the response.

The result of the commands are tallied and returned in an xml with as root the `response` tag.

The body of the xml contains either an error tag (when connecting to MMBase failed), or a list of tags containing the results of the command passed, in the order that those commands were given. The response does not contain any security information, even if the request did.

```
<!ELEMENT response (getdata|getnew|getnewrelation|getrelations|getconstraints|g
```

Each command contains the result of that command or an `error` tag if the command failed.

## Note

If Dove cannot handle the command, an `error` tag is returned instead to stay compliant with the response dtd. It is recommended you validate a document before passing it to the Dove.

### How to Call Dove

The `org.mmbase.applications.dove.Dove` Java class contains the code that processes the xml tree. If you want to use the Dove from java code, you should first create a DOM tree, containing the command to run. You then instantiate Dove, using a DOM Document to be used for creating the response, and finally call the `executeRequest` method, passing the entity that contains the root of the request. For more information on calling the class we refer to the MMBase API documentation

If you want to call Dove remote, you can also install the DoveServlet (comments on configuring this servlet can be found in the sample web.xml that comes with MMBase). You can call this servlet using a html form containing an 'xml' field that should then contain the xml to parse and run. The result of the call is the xml response (in text format).

## Security

It is possible to pass a `security` tag to Dove, enabling the class to log in on MMBase. The tag accepts a `name` and `password` attribute, as well as a `cloud` attribute containing the name and a `method` attribute containing the login method name. These attributes are similar in meaning to the attributes of the `cloud` tag in the MMBase taglib (documented elsewhere). All attributes are optional. In general, `name` and `password` will be required unless the default method for authentication is overruled.

```
<!ELEMENT security EMPTY>
<!ATTLIST security name NMOKEN #IMPLIED >
<!ATTLIST security password NMOKEN #IMPLIED >
<!ATTLIST security method CDATA 'name/password' >
<!ATTLIST security cloud CDATA 'mmbase' >
```

### Example 1. Example:

```
<request>
  <security name="admin" password="admin2k" />
  <getdata><object number="675" /></getdata>
</request>
```

## Note

Dove does not allow anonymous access: if you do not provide a security login, you need to provide a secure MMCI cloud as an additional parameter when passing the XML.

## Errors

When the Dove fails to execute a command it returns an `error` tag. An error tag contains a `type` attribute that details the type of error, as follows:

- **parser.** The xml given is invalid or does not follow the grammar. This likely means there is a bug in the client code.
- **server.** The code invoked is either not yet implemented or another, server-related, error occurred (such as no memory, bad configuration, etc.). Server errors entirely fail a request.
- **client.** The data requested could not be retrieved or values specified were invalid. I.e. a requested node does not exist (any more), or a put failed due to invalid data.

The body of the tag contains a description of the actual error.

```
<!ENTITY % errortype (server,parser,client)>
<!ELEMENT error (#PCDATA)>
<!-- ATTENTION: error type %errortype #REQUIRED -->
```

If the error occurs due to a security issue, only the error is returned (no command is executed).

### Example 2. Example:

```
<response>
  <error type="client">Authentication error : password invalid</error>
</response>
```

If the error occurred while attempting to execute a command, it occurs at the location where the error occurred.

### Example 3. Example:

```
<response>
  <getdata>
    <object number="234">
      <error type="client">node not found</error>
    </object>
  </getdata>
</response>
```

### Note

Except during a put, Dove attempts to continue execution when it runs into an error.

## Commands

The following pages list the commands that can be passed to the Dove.

An implementation should support the following commands:

## Dove Commands

- **getdata.** Retrieves an MMBase node
- **getnew.** Returns a new, empty node
- **getnewrelation.** Returns a new, empty relation node
- **getrelations.** Returns the relations of a node of a certain role
- **getconstraints.** Returns the constraints of a node type
- **getlist.** Returns a list of nodes following a constraint
- **put.** Passes changes to make to MMBase

## getdata

A getdata command retrieves objects from MMBase by their number.

The command should contain one or more `object` tags, each tag referencing an object through the `number` attribute. If the referenced object can be found, the object, along with type and all its fields, is returned in the response. If retrieving the object fails, the returned response `object` tag contains an `error` tag describing the problem.

If you do not want to retrieve all the object's fields, you can specify the fields you want to retrieve by including `field` tags in the body of the `object` tag. The field tags' `name` attribute identifies the fields you want to retrieve. If at least one field tag is given, only the selected fields are returned. Note that you should not retrieve relation objects directly - use `getrelations` instead.

```
<!ELEMENT getdata (object+) >
<!ATTLIST getdata id ID #IMPLIED>

<!ELEMENT object (field*)>
<!ATTLIST object number NMTOKEN #REQUIRED >

<!ELEMENT field EMPTY>
<!ATTLIST field name NMTOKEN #REQUIRED >
```

The response contains, aside from the fields, the type of the object.

```
<!ELEMENT getdata (object+) >
<!ATTLIST getdata id ID #IMPLIED>

<!ELEMENT object (field*)>
<!ATTLIST object did ID #IMPLIED>
<!ATTLIST object number NMTOKEN #REQUIRED >
<!ATTLIST object type NMTOKEN #REQUIRED >

<!ELEMENT field (#PCDATA) >
<!ATTLIST field did ID #IMPLIED>
<!ATTLIST field name NMTOKEN #REQUIRED >
```

### Example 4. Example

```
<request>
  <getdata id="1">
    <object number="675" />
  </getdata>

  <getdata id="2">
    <object number="672" >
      <field name="title" />
    </object>
    <object number="671" >
      <field name="name" />
      <field name="url" />
    </object>
  </getdata>
</request>
```

The result of this command could look like this:

```
<response>
  <getdata id="1">
    <object number="675" type="people">
      <field name="firstname">Harrison</field>
      <field name="lastname">Ford</field>
      <field name="email">harrisonford@hollywood.com</field>
    </object>
  </getdata>

  <getdata id="2">
    <object number="672" type="movies">
      <field name="title">Star Wars</field>
    </object>
    <object number="671" type="urls">
      <field name="name" >Star Wars Site</field>
      <field name="url" >http://www.starwars.com</field>
    </object>
  </getdata>
</response>
```

## getrelations

A `getrelations` command obtains the relation objects belonging to the specified objects, filtered by a specified destination type or relation role.

The command should contain one or more `object` tags, each tag referencing an object through the `number` attribute. If the referenced object can be found, the object's relations are retrieved and returned.

An `object` tag may include one or more `relation` tags, allowing to retrieve a narrow selection of relations, relating to a specified type of object or using a specified role. Restricting the selection is done by supplying a `role` or a `destinationtype` attribute (or both) to a relation tag.

By default, all fields of a relation are returned. If you only want to retrieve certain fields, you can do so by specifying these fields by including `field` tags in the body of the relation tag. The field tags' `name` attribute identifies the fields you want to retrieve. If at least one field tag is given, only the se-

lected fields are returned.

## Note

In Dove 1.0, a `searchdir` attribute can be specified to limit the result based on directionality. The value 'source' limits the returned relations to those that link TO the specified object FROM a 'source' object, 'destination' limits the relations to those that link FROM the specified object TO a 'destination' object, and 'both' (the default) returns all relations, except those that would violate directionality (i.e. uni-directional relations).

```
<!ENTITY % searchdir "(source,destination,both)">

<!ELEMENT getrelations (object+) >
<!ATTLIST getrelations id ID #IMPLIED>

<!ELEMENT object (relation*)>
<!ATTLIST object number NMTOKEN #REQUIRED >

<!ELEMENT relation (field*)>
<!ATTLIST relation role NMTOKEN #IMPLIED >
<!ATTLIST relation searchdir %searchdir #IMPLIED >
<!ATTLIST relation destinationtype NMTOKEN #IMPLIED >

<!ELEMENT field (#PCDATA) >
<!ATTLIST field name NMTOKEN #REQUIRED >
```

The response contains the requested relation objects, wrapped by the object used to retrieve them. All additional fields of the relations are also returned.

## Note

The common relation fields `snumber`, `dnumber` and `rnumber` are not returned as field elements, but as the attributes `source`, `destination`, and `role`. These fields contain MMBase object numbers (`snumber` and `dnumber` of the relation). It is possible to check if the initiating node was the source or the destination of the relations, by checking whether its number is in the 'source' or in the 'destination' attribute. The `role` attribute contains the forward role (`sname` field) of the relation.

```
<!ELEMENT getrelations (object+) >
<!ATTLIST getrelations id ID #IMPLIED>

<!ELEMENT object (relation*)>
<!ATTLIST object number NMTOKEN #REQUIRED >

<!ELEMENT relation (field*)>
<!ATTLIST relation number NMTOKEN #REQUIRED >
<!ATTLIST relation type NMTOKEN #REQUIRED >
<!ATTLIST relation role NMTOKEN #REQUIRED >
<!ATTLIST relation source NMTOKEN #REQUIRED >
<!ATTLIST relation destination NMTOKEN #REQUIRED >

<!ELEMENT field (#PCDATA) >
<!ATTLIST field name NMTOKEN #REQUIRED >
```

## Example 5. Example:

```
<request>
  <getrelations id="1">
```

```
<object number="675" />
</getrelations>

<getrelations id="2">
  <object number="672" >
    <relation role="part" />
    <relation destinationtype="scenes" role="posrel">
      <field name="pos" />
    </relation>
  </object>
</getrelations>

</request>
```

Which may result in the following response:

```
<response>

  <getrelations id="1">
    <object number="675" type="people">
      <relation number="682" type="insrel" source="675" destination="681" role=
      <relation number="684" type="insrel" source="675" destination="683" role=
      <relation number="690" type="insrel" source="675" destination="678" role=
    </object>
  </getrelations>

  <getrelations id="2">
    <object number="672" type="movies">
      <relation number="677" type="insrel" source="672" destination="678" role=
      <relation number="700" type="posrel" source="672" destination="699" role=
        <field name="pos" >1</field>
      </relation>
      <relation number="702" type="posrel" source="672" destination="701" role=
        <field name="pos" >2</field>
      </relation>
      <relation number="704" type="posrel" source="672" destination="703" role=
        <field name="pos" >3</field>
      </relation>
    </object>
  </getrelations>

</response>
```

## Nesting calls (getdata/getrelations)

### Note

Deep nesting is possible only with Dove version 1.0 that is available with MMBase 1.5.1 and up.

Loading one object or set of relations at a time can be time consuming, especially if a lot of inter-related objects need to be loaded.

It is possible to define a load action of related objects in one command, by nesting objects and their relations.

The nested request should follow this DTD:

```
<!ELEMENT object ((field|relation)*)>
```

```
<!ATTLIST object number NMTOKEN #IMPLIED >

<!ELEMENT relation ((field|object)*)>
<!ATTLIST relation searchdir %searchdir #IMPLIED >
<!ATTLIST relation role NMTOKEN #IMPLIED >
<!ATTLIST relation destinationtype NMTOKEN #REQUIRED >
```

and the nested response should follow this one:

```
<!ELEMENT object ((field|relation)*)>
<!ATTLIST object number NMTOKEN #REQUIRED >
<!ATTLIST object type NMTOKEN #REQUIRED >

<!ELEMENT relation ((field|object)*)>
<!ATTLIST relation number NMTOKEN #REQUIRED >
<!ATTLIST relation type NMTOKEN #REQUIRED >
<!ATTLIST relation role NMTOKEN #REQUIRED>
<!ATTLIST relation source NMTOKEN #REQUIRED >
<!ATTLIST relation destination NMTOKEN #REQUIRED >
```

## Example 6. Example

A possible combination of nesting objects and relations could look like this:

```
<request>
<getdata id="1">
  <object number="672">

    <field name="title" />

    <relation role="part">
      <object >
        <field name="name" />
        <relation destinationtype="people" role="actor" >
          <object >
            <field name="firstname" />
            <field name="lastname" />
          </object>
        </relation>
      </object>
    </relation>

    <relation destinationtype="scenes" role="posrel">
      <field name="pos" />
      <object>
        <field name="title" />
      </object>
    </relation>

  </object>

</getdata>
</request>
```

And this could be the response:

```
<response>

  <getdata id="1">
    <object number="672" type="movies">

      <field name="title" >
        <relation number="677" type="insrel" source="672" destination="678" role=
          <object number="678" type="elements">
            <field name="name" >Han Solo</field>
            <relation number="690" type="insrel" source="678" destination="675" r
              <object number="690" type="people" >
                <field name="firstname" >Harrison</field>
                <field name="lastname" >Ford</field>
              </object>
            </relation>
          </object>
        </relation>

        <relation number="700" type="posrel" source="672" destination="699" role=
          <field name="pos" >1</field>
          <object number="699" type="scenes">
            <field name="title">Entering Mos Eisly</field>
          </object>
        </relation>

        <relation number="702" type="posrel" source="672" destination="701" role=
          <field name="pos" >2</field>
          <object number="701" type="scenes">
            <field name="title">These are not the droids you are looking for</fie
          </object>
        </relation>

        <relation number="704" type="posrel" source="672" destination="703" role=
          <field name="pos" >3</field>
          <object number="703" type="scenes">
            <field name="title">Boring conversation anyway</field>
          </object>
        </relation>

      </object>
    </getdata>
  </response>
```

## getnew

A `getnew` command creates a new, empty object of a specified object type.

The new object is returned with a (system-generated) temporary object number, and any fields for this object, with default values according to the system. This call is for generating normal objects - for generating relations use `getnewrelation`.

The command requires a `type` attribute that holds the type to generate

```
<!ELEMENT getnew EMPTY >
<!ATTLIST getnew id ID #IMPLIED>
<!ATTLIST getnew type NMTOKEN #REQUIRED>
```

The response contains the new object, in a fashion similar to `getdata`. The `getnew` element itself also contains the original type passed. All fields of the object are returned.

```
<!ELEMENT getnew (object|error) >
<!ATTLIST getnew id ID #IMPLIED>
```

```
<!ATTLIST getnew type NMTOKEN #REQUIRED>

<!ELEMENT object (field*)>
<!ATTLIST object number NMTOKEN #REQUIRED >
<!ATTLIST object type NMTOKEN #REQUIRED >

<!ELEMENT field (#PCDATA) >
<!ATTLIST field name NMTOKEN #REQUIRED >
```

### Example 7. Example:

```
<request>
  <getnew type="people" />
</request>
```

And the result may look like this:

```
<response>

  <getnew id="1" type="people">
    <object number="n62967893564" type="people">
      <field name="firstname" />
      <field name="lastname" />
      <field name="email" />
    </object>
  </getnew>

</response>
```

## getnewrelation

A `getnewrelation` command creates a new, empty relation object with a specified role, source, and destination.

The new object is returned with a (system-generated) temporary object number, role, source, and destination, and any additional fields for this object, with default values according to the system. This call is for generating relations - for normal objects use `getnew`.

The command requires a role, source, and destination attribute that holds information for the relation to generate.

```
<!ELEMENT getnewrelation EMPTY>
<!ATTLIST getnewrelation id ID #IMPLIED>
<!ATTLIST getnewrelation role NMTOKEN #REQUIRED >
<!ATTLIST getnewrelation source NMTOKEN #REQUIRED >
<!ATTLIST getnewrelation destination NMTOKEN #REQUIRED >
```

The response contains the new relation, in a fashion similar to `getrelations`. The `getnewrelation` element itself also contains the original attributes passed. All additional fields of the relation are returned.

### Note

`snumber`, `dnumber` and `rnumber` are not returned as fields, but as the attributes `source`, `destination`, and `role`.

```
<!ELEMENT getnewrelation (relation|error)>
<!--ATTLIST getnewrelation id ID #IMPLIED-->
<!--ATTLIST getnewrelation role NMOKEN #REQUIRED -->
<!--ATTLIST getnewrelation source NMOKEN #REQUIRED -->
<!--ATTLIST getnewrelation destination NMOKEN #REQUIRED -->

<!--ELEMENT relation (field*)>
<!--ATTLIST relation number NMOKEN #REQUIRED -->
<!--ATTLIST relation type NMOKEN #REQUIRED -->
<!--ATTLIST relation role NMOKEN #REQUIRED -->
<!--ATTLIST relation source NMOKEN #REQUIRED -->
<!--ATTLIST relation destination NMOKEN #REQUIRED -->

<!--ELEMENT field (#PCDATA) -->
<!--ATTLIST field name NMOKEN #REQUIRED -->
```

### Example 8. Example:

```
<request>
  <getnewrelation role="posrel" source="134" destination="n72647968" />
</request>
```

And the result:

```
<response>

  <getnewrelation role="posrel" source="134" destination="n72647968" />
    <relation number="n62967893568" role="posrel" type="posrel"
      source="134" destination="n72647968" >
      <field name="pos" >0</field>
    </relation>
  </getnewrelation>

</response>
```

## getconstraints

A `getconstraints` command obtains constraint information for an object type.

The type should be specified in the `type` attribute. An optional `xml:lang` attribute can be specified to indicate the preferred language of any gui information (help text, prompts, etc.). A Dove implementation that supports this will return data in the preferred language where possible. the language specified should be in the ISO 639-1 [<http://lcweb.loc.gov/standards/iso639-2/>] format, such as 'en', 'nl', or the like.

```
<!--ELEMENT getconstraints (object+) -->
<!--ATTLIST getconstraints id ID #IMPLIED-->
<!--ATTLIST getconstraints type NMOKEN #REQUIRED-->
<!--ATTLIST getconstraints xml:lang NMOKEN "en"-->
```

The response contains various tags containing basic information about the object type, the fields of the type, and any allowed relations.

Tags supported in API 1.0 are `singularname`, `pluralname`, and `description`.

The `fields` tag lists all editable fields of an object. Each `field` has its own tag, with a `name` attribute and a number of subtags: `guiname`, `guitype`, `maxlength` and `required`.

The `guitype` of a field has a mime-type like format of the form `datatype/presentationtype`.

The `datatype` part is one of `string`, `int`, `long`, `float`, `double`, `datetime`, or `binary`.

The `presentationtype` may vary (common types are `line`, `text`, `date`, or `image`).

## Note

The `description` subtag of a field is available only in version 1.0 of the Dove.

```
<!ELEMENT getconstraints (error|(singularname,pluralname,description,fields,relations))>
<!ATTLIST getconstraints id ID #IMPLIED>
<!ATTLIST getconstraints type NMTOKEN #REQUIRED >
<!ATTLIST getconstraints xml:lang NMTOKEN "en" >

<!ELEMENT singularname (#PCDATA) >
<!ATTLIST singularname xml:lang NMTOKEN "en" >

<!ELEMENT pluralname (#PCDATA) >
<!ATTLIST pluralname xml:lang NMTOKEN "en" >

<!ELEMENT description (#PCDATA) >
<!ATTLIST description xml:lang NMTOKEN "en" >

<!ELEMENT fields (field*) >

<!ELEMENT relations (relation*) >

<!ELEMENT relation EMPTY >
<!ATTLIST relation role NMTOKEN #REQUIRED>
<!ATTLIST relation destinationtype NMTOKEN #REQUIRED >

<!ELEMENT field (guiname,description,guitype,maxlength,required)>
<!ATTLIST field name NMTOKEN #REQUIRED >

<!ELEMENT guiname (#PCDATA) >
<!ATTLIST guiname xml:lang NMTOKEN "en" >

<!ELEMENT guitype (#PCDATA) >

<!ELEMENT maxlength (#PCDATA) >

<!ELEMENT required (#PCDATA) >
```

## Example 9. Example:

```
<request>
  <getconstraints type="people" xml:lang="nl"/>
</request>
```

And a possible result:

```
<response>
  <getconstraints type="people" xml:lang="nl">
```

```
<singularname xml:lang="nl">Persoon</singularname>
<pluralname xml:lang="nl">Personen</pluralname>
<description xml:lang="nl">Persoonsgegevens.</description>
<fields>
  <field name="firstname">
    <guiname xml:lang="nl">Voornaam</guiname>
    <description>Voornaam van de persoon</description>
    <guitype>string/line</guitype>
    <maxlength>20</maxlength>
    <required>FALSE</required>
  </field>
  <field name="lastname">
    <guiname xml:lang="nl">Achternaam</guiname>
    <description>Achternaam van de persoon</description>
    <guitype>string/line</guitype>
    <maxlength>50</maxlength>
    <required>TRUE</required>
  </field>
  <field name="email">
    <guiname xml:lang="nl">Email adres</guiname>
    <description>Email adres van de persoon</description>
    <guitype>string/line</guitype>
    <maxlength>35</maxlength>
    <required>FALSE</required>
  </field>
</fields>
<relations>
  <relation destinationtype="images" role="related" />
  <relation destinationtype="movies" role="actor" />
</relations>
</getconstraints>

</response>
```

## getlist

The `getlist` command is used to retrieve lists of nodes using a set of criteria, such as nodetype and field content. The tag contains a set of query tags, each specifying the constraints to retrieve nodes with.

Each query element should have a `xpath` attribute to indicate the nodemanager to run the list on, and optionally a `where` attribute that contains the field-constraints.

The `xpath` value is a simplified xpath of the format `"/*@nodemanagername"`, such as `"/*@people"`. To do a multilevel query, make a list of nodemanagers separated by `"/"` such as `"/*@people/roles"`. It is possible that in future implementations more versatile xpaths can be used.

The `where` attribute uses a SQL syntax to check a field, such as `"name LIKE '%a%'"`. It is in all respects the same as the `constraints` attribute as used in the `mm:listnodes` tag of the MMBase taglib (described elsewhere).

Fields to retrieve can be limited in the same way as with a `getdata` command: by specifying the fields in an `object` tag. Note that when doing a multilevel query, you must specify an `object` tag with at least one field.

```
<!ELEMENT getlist (query+) >
<!ATTLIST getlist id ID #IMPLIED>

<!ELEMENT query (object?)>
<!ATTLIST query xpath CDATA #REQUIRED >
<!ATTLIST query where CDATA #IMPLIED >

<!ELEMENT object (field*)>
```

```
<!ELEMENT field EMPTY>
<!ATTLIST field name NMTOKEN #REQUIRED >
```

The response contains a list of the query tags, each filled with object tags (and their field tags) that follow the constraints.

```
<!ELEMENT getlist (query+|error) >
<!ATTLIST getlist id ID #IMPLIED>

<!ELEMENT query (object?|error)>
<!ATTLIST query xpath CDATA #REQUIRED >
<!ATTLIST query where CDATA #IMPLIED >

<!ELEMENT object ((field|relation)*)>
<!ATTLIST object number NMTOKEN #REQUIRED >
<!ATTLIST object type NMTOKEN #REQUIRED >

<!ELEMENT field (#PCDATA)>
<!ATTLIST field name NMTOKEN #REQUIRED >
```

### Example 10. Example:

```
<request>

  <getlist id="7">
    <query xpath="/*@people" where="gender ='M'">
      <object>
        <field name="firstname" />
        <field name="lastname" />
        <field name="email" />
      </object>
    </query>
  </getlist>

</request>
```

The result of this query could look something like this:

```
<response>

  <getlist id="7">
    <query xpath="/*@people" where="firstname LIKE '%a%'">
      <object number="675" type="people">
        <field name="firstname">Harrison</field>
        <field name="lastname">Ford</field>
        <field name="email">harrisonford@hollywood.com</field>
      </object>
      <object number="234" type="people">
        <field name="firstname">Mark</field>
        <field name="lastname">Hammil</field>
        <field name="email">mark12@yahoo.com</field>
      </object>
      <object number="296" type="people">
        <field name="firstname">Carry</field>
        <field name="lastname">Fisher</field>
        <field name="email">cfish@yahoo.com</field>
      </object>
    </query>

  </getlist>
```

```
</getlist>
</response>
```

**Example 11. Example with a multilevel query:**

```
<request>
  <getlist id="7">
    <query xpath="/*@people/roles" where="people.gender ='M' ">
      <object>
        <field name="people.firstname" />
        <field name="role.description" />
      </object>
    </query>
  </getlist>
</request>
```

The result of this query could look something like this:

```
<response>
  <getlist id="7">
    <query xpath="/*@people/roles" where="people.gender ='M' ">
      <object number="675" type="virtualnodes_1085647783628">
        <field name="people.firstname">Harrison</field>
        <field name="role.description">Actor</field>
      </object>
    </query>
  </getlist>
</response>
```

## put

The put command is used to update MMBase by sending a set of nodes and relations to add or change. Each put command contains two node lists: the original tag contains a set of (original) objects that should be changed or deleted, the new tag contains a corresponding set of object with the needed changes, as well as a set of objects or relations that are new and need to be added.

### Note

Changes made by the put command are created through a transaction. This means that if an error occurs (i.e. because changes were invalid), nothing is changed, maintaining the consistency of the system. This same mechanism also means that large updates through Dove are discouraged as they are quite slow and eat resources.

The object and relation tags in original follow the general format (specifying a number attribute and field tags) and should be existing nodes - the contents of the fields specified in this list should match data in the database. If data is specified that is incorrect, an error is generated by the system, dictating an invalid attempt to change the data

### Note

This check is used as a way to prevent edits from concurrent users from crossing each oth-

er: If two users attempt changes on the same object at the same time, one of them will fail, which ensures that data does not become corrupted or invalid.

Each object or relation in the original list should have a `status` attribute which indicates what should happen to the object: 'change' (edit content) or 'delete' (remove the object from the cloud).

The new tag lists all objects that should either be changed or added - objects with a status of 'delete' should not be included in this list. New objects in this list should have the status 'new'. All data mentioned here is changed or added - note that if a field is not mentioned, the content of that field (either default or existing content) is unchanged.

```
<!ENTITY % putstatus "(change,delete,new)">

<!ELEMENT put (original,new) >
<!ATTLIST put id ID #IMPLIED>

<!ELEMENT original ((object|relation)*) >
<!ELEMENT new ((object|relation)*) >

<!ELEMENT object ((field|relation)*)>
<!ATTLIST object number NMTOKEN #IMPLIED >
<!ATTLIST object status %putstatus #IMPLIED>

<!ELEMENT relation ((field|object)*)>
<!ATTLIST relation role NMTOKEN #IMPLIED >
<!ATTLIST relation searchdir %searchdir #IMPLIED >
<!ATTLIST relation destinationtype NMTOKEN #REQUIRED >
<!ATTLIST relation status %putstatus #IMPLIED>

<!ELEMENT field (#PCDATA)>
<!ATTLIST field name NMTOKEN #REQUIRED >
<!ATTLIST field href CDATA #IMPLIED >
```

If the out is successful, the response tag contains a result list of object and relation tags in the new tag.

This list consists of all changed and added nodes - deleted nodes are not included.

```
<!ELEMENT put (new|error) >
<!ATTLIST put id ID #IMPLIED>

<!ELEMENT new ((object|relation)*) >

<!ELEMENT object ((field|relation)*)>
<!ATTLIST object number NMTOKEN #REQUIRED >
<!ATTLIST object type NMTOKEN #REQUIRED >

<!ELEMENT relation ((field|object)*)>
<!ATTLIST relation number NMTOKEN #REQUIRED >
<!ATTLIST relation type NMTOKEN #REQUIRED >
<!ATTLIST relation role NMTOKEN #REQUIRED>
<!ATTLIST relation source NMTOKEN #REQUIRED >
<!ATTLIST relation destination NMTOKEN #REQUIRED >

<!ELEMENT field (#PCDATA)>
<!ATTLIST field name NMTOKEN #REQUIRED >
```

## Example 12. Example:

```
<request>
  <put>
    <original>
      <object number="675" status="change">
        <field name="firstname">Harrison</field>
        <field name="lastname">Ford</field>
        <field name="email">harrisonford@hollywood.com</field>
      </object>
      <object number="296" status="delete">
        <field name="firstname">Carry</field>
        <field name="lastname">Fisher</field>
        <field name="email">cfish@yahoo.com</field>
      </object>
    </original>
    <new>
      <object number="675">
        <field name="email">harrison@aol.com</field>
      </object>
      <object status="new">
        <field name="firstname">Alec</field>
        <field name="lastname">Guinness</field>
        <field name="email">n.a.</field>
      </object>
    </new>
  </put>
</request>
```

```
<response>
  <put>
    <new>
      <object number="675">
        <field name="firstname">Harrison</field>
        <field name="lastname">Ford</field>
        <field name="email">harrison@aol.com</field>
      </object>
      <object number="1423">
        <field name="firstname">Alec</field>
        <field name="lastname">Guinness</field>
        <field name="email">n.a.</field>
      </object>
    </new>
  </put>
</response>
```