Simon Schwantzer (IMC) - 2 October 2013

MIRROR Spaces Service

Contents

Installation

This version of the MIRROR Spaces Service requires Openfire 3.8.x. To install the plugin perform the following steps:

1. Open the administration console of Openfire and click on the "Plugins" tab.
2. In the "Upload Plugin" section of the page, select the spacesService.jar file and submit it by pressing on the "Upload Plugin" button.
3. After a few seconds, new entry "MIRROR Spaces Service" should show up in the plugins list. You can validate the installation by opening the info log ("Server" > "Server Manager" > "Logs" > "Info") , which should contain a line:
   `MIRROR Spaces plugin initialized.`
4. Open the new "MIRROR Spaces" tab and click on "Settings" to open the general settings for the service.
5. Check the entries in the "Server Configuration" box. Update the settings if necessary.

Updating an Existing Installation

This is the first version of the plugin for Openfire 3.8.x. If you want to upgrade an older version of Openfire, perform the following steps:

1. Open the "Plugins" tab of the openfire administration console and delete the old "MIRROR Spaces Service".
   Note: Spaces and stored data will remain unaffected.
2. Upgrade your openfire installation as described in the upgrade guide.
3. Deploy the plugin as described above.

Configuration

If you also deployed the MIRROR Persistence Service you need to connect the services in order to work properly:

1. Open the new "MIRROR Spaces" tab and click on "Settings" to open the general settings for the service.
2. Select the checkbox "Connect to the MIRROR Persistence Service." in the persistence settings.
3. Submit the change by pressing the "Save" button

Introduction

The MIRROR Spaces Service is the server-side implementation of the Interoperability Framework as described in the D2.2 Space Concept. This document describes how to use the service to manage and discover spaces.

To programmatically ensure the MIRROR spaces service is available, it is possible to send a service discovery request to the openfire server. The request looks like this:

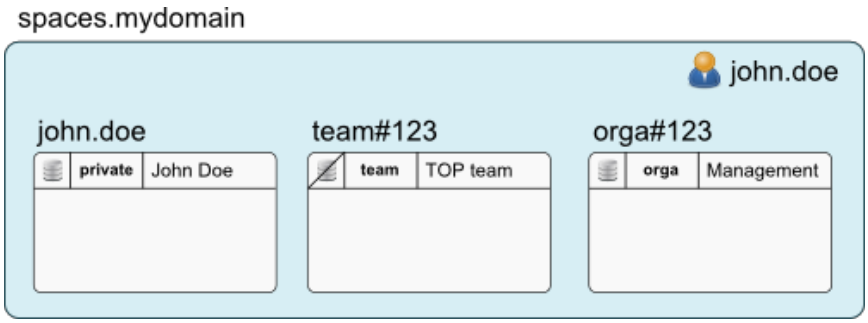
```
<iq id="info1"
    to="mydomain"
    from="john.doe@mydomain/client"
    type="get">
  <query xmlns="http://jabber.org/protocol/disco#items" />
</iq>
```

If the service is available, an item with the name “MIRROR spaces service” with the related service domain will be listed.

```
<iq id="info1"
    to="john.doe@mydomain/client"
    from="mydomain"
    type="result">
  <query xmlns="http://jabber.org/protocol/disco#items">
    [...]
    <item jid="spaces.mydomain" name="MIRROR spaces service"/>
    [...]
  </query>
</iq>
```

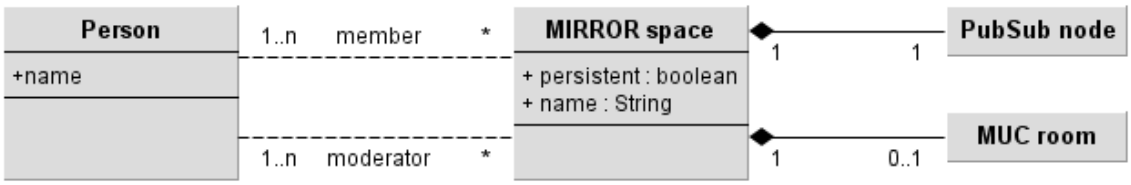
Space Structure and User Management

The MIRROR spaces service provides an interface to manage and discover *spaces*. The service is user-centric, i.e., only the spaces the requesting user is member of are visible for him.

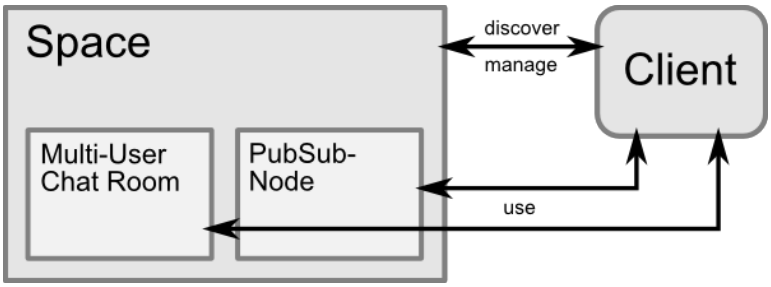


A space itself consists of some metadata and some *channels*. Currently available channels are:

- a data exchange channel, provided by a publish-subscribe node
- a communication channel in form of a multi-user chat room[1]



The channels are managed by the spaces service, i.e., the channel configuration is derived from the space configuration and will be automatically updated when a space is modified.



The space metadata contains the following fields:

id	A unique space identifier. This identifier is set and returned by the spaces service during space creation.
type	Type of the space, i.e., `private`, `team` or `orga`.
name	A display name of the space. This name can be configured by the client. In the default configuration, the name equals the space id.
persistent	The persistence field defines the lifetime of data objects published on a space and has the following options: • `true`: Data objects published on the space is stored without expiration date.

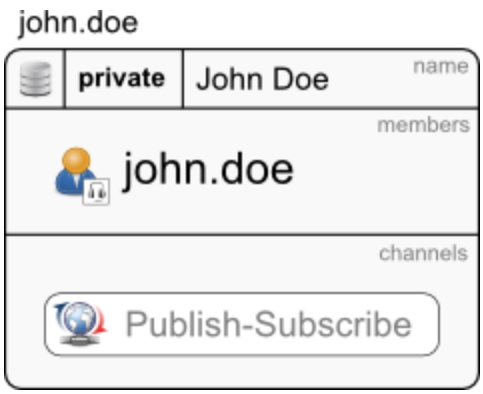
[1] The communication channel is not available in private spaces.

	• `false`: Data objects published on the space are not stored. • XSD duration string: A duration for which data objects lasts until the are purged automatically.[2]

Whilst private spaces have one and only one member, team and organizational spaces may have multiple members. The list of members is provided by the client during the space creation and can later on be changed via the space configuration.
Access control is provided by a simple role mechanism. To configure a space, a user must be “moderator” of the space. By default, only the space creator is categorized as moderator. Moderators may grant this role to other team members.
Note: A space can have multiple moderators.

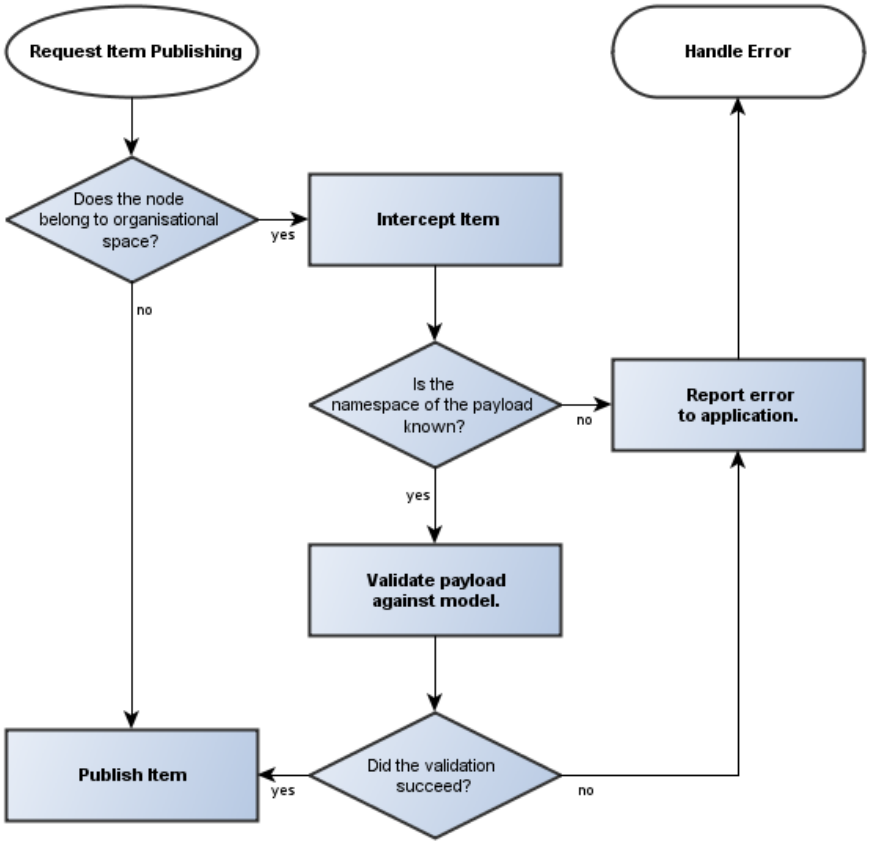
members	List of members of the space. Members may read the metadata and channel information.
moderators	A subset of members, which are allowed to configure the space and manage the members.

The following visualisations represents two exemplary spaces, which are used in the following sections. The first one (“john.doe”) is the private and persistent space of John Doe. The second one (“team#123”) is a non-persistent team space with John Doe and Jane Doe as members. John acts as moderator of this space.



Organizational spaces also has an additional list of supported data models. If a user tries to publish a data object on the publish subscribe channel provided by a organizational space, the spaces service will intercept the package, check if the data model is unlocked by using the namespace, and validates the data object against the related data model schema. Only if all these steps succeed, the data object will be published.

[2] Durations are available since version 0.5 of the MIRROR Spaces Service.



Space Discovery

The MIRROR spaces service can be discovered via the service discovery mechanism described in XEP-0030. The service handles three types of service requests:

1. Service information requests (`#info`) to the service itself.
2. Service items requests (`#items`) to the service itself.
3. Service information request (`#info`) to specific spaces managed by the service.

A service information query without a node attribute returns the features provided by the MIRROR spaces service.

Request:

```
<iq type="get"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="info2">
  <query xmlns="http://jabber.org/protocol/disco#info"/>
</iq>
```

Response:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="info2">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <identity
        category="component"
        type="misc"
        name="MIRROR spaces service"/>
    <feature var="http://jabber.org/protocol/disco#info"/>
    <feature var="http://jabber.org/protocol/disco#items"/>
    <feature var="urn:xmpp:spaces"/>
  </query>
</iq>
```

An service items query without a `node` attribute will list all spaces managed by the service. Each item has the same JID (which equals the service domain), a display name and a node identifier.
Note: The node identifier equals the space identifier.

Request:

```
<iq type="get"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="items1">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>
```

Response:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="items1">
  <query xmlns="http://jabber.org/protocol/disco#items">
    <item jid="spaces.mydomain"
          node="john.doe"
          name="John Doe"/>
    <item jid="spaces.mydomain"
          node="team#123"
          name="TOP team"/>
    <item jid="spaces.mydomain"
          node="orga#123"
          name="Management Department"/>
  </query>
</iq>
```

With the previous request, we retrieve the identifiers of three spaces: `john.doe`, `test#123` and `orga#123`. To retrieve further information about a space, it is possible to send an node

info query. This query will list the space"s metadata and members.

Request:

```
<iq type="get"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="info3">
  <query xmlns="http://jabber.org/protocol/disco#info" node="team#123"/>
</iq>
```

Response:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="info3">
  <query xmlns="http://jabber.org/protocol/disco#info" node="team#123">
    <identity category="spaces" type="space" name="TOP team"/>
    <feature var="http://jabber.org/protocol/disco#info"/>
    <feature var="urn:xmpp:spaces">
    <x xmlns="jabber:x:data" type="result">
      <field var="FORM_TYPE" type="hidden">
        <value>urn:xmpp:spaces:metadata</value>
      </field>
      <field var="spaces#type" type="hidden">
        <value>team</value>
      </field>
      <field var="spaces#persistent" label="Persistent" type="boolean">
        <value>false</value>
      </field>
      <field var="spaces#name" label="Name" type="text-single">
        <value>TOP team</value>
      </field>
      <field var="spaces#members" label="Members" type="jid-multi">
        <value>john.doe@mydomain</value>
        <value>jane.doe@mydomain</value>
      </field>
      <field var="spaces#moderators" label="Moderators" type="jid-multi">
        <value>john.doe@mydomain</value>
      </field>
    </x>
  </query>
</iq>
```

The space request in the previous example is therefore a persistent team space called “TOP team”.

Space Management

Spaces can be created, modified and deleted with IQ stanzas addressed to the service. The IQ child element is named `spaces` and contains one or more command elements. Each request is, as defined in the XMPP reference, responded by either an result or an error.

Create a Space

To create a space, the command element is named `create`. The space configuration is set by a following `configure` element.[3] It is possible to omit the configure element. In this case, a private space with default settings is created for the requesting user.

Request (private space with default settings):

```
<iq type="set"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="create1">
  <spaces xmlns="urn:xmpp:spaces">
    <create/>
  </spaces>
</iq>
```

Request (space with individual settings):

```
<iq type="set"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="create2">
  <spaces xmlns="urn:xmpp:spaces">
    <create/>
    <configure>
      <x xmlns="jabber:x:data" type="submit">
        <field var="FORM_TYPE" type="hidden">
          <value>
            urn:xmpp:spaces:config
          </value>
        </field>
        <field var="spaces#type">
          <value>team</value>
        </field>
        <field var="spaces#persistent">
          <value>false</value>
        </field>
        <field var="spaces#name">
          <value>TOP team</value>
        </field>
        <field var="spaces#members">
          <value>john.doe@mydomain</value>
```

[3] Be aware that the `configure` element is direct child of the `spaces` element, not of the `create` element.

```
          <value>jane.doe@mydomain</value>
        </field>
        <field var="spaces#moderators">
          <value>john.doe@mydomain</value>
        </field>
      </x>
    </configure>
  </spaces>
</iq>
```

Response on success:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="create2">
  <spaces xmlns="urn:xmpp:spaces">
    <create space="team#123"/>
  </spaces>
</iq>
```

In this example, the creation succeeded. The space identifier attached as attribute to the `create` element. Errors are handled with XMPP error responses as defined in the XMPP core reference. Possible conditions are:

`conflict`	A private space cannot be created because it already exists.
`bad_request`	The configuration is corrupt.

Configure a Space

To configure a space, a single `configure` element is attached to the `spaces` element. The identifier of the space to configure is provided with the attribute `space` of the configure element.

Request:

```
<iq type="set"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="configure1">
  <spaces xmlns="urn:xmpp:spaces">
    <configure space="team#123">
      <x xmlns="jabber:x:data" type="submit">
        <field var="FORM_TYPE">
          <value>
            urn:xmpp:spaces:config
          </value>
        </field>
```

```
        <field var="spaces#type">
          <value>team</value>
        </field>
        <field var="spaces#persistent">
          <value>false</value>
        </field>
        <field var="spaces#name">
          <value>TOP team with monster</value>
        </field>
        <field var="spaces#members">
          <value>john.doe@mydomain</value>
          <value>jane.doe@mydomain</value>
          <value>cookie.monster@mydomain</value>
        </field>
        <field var="spaces#moderators">
          <value>john.doe@mydomain</value>
        </field>
      </x>
    </configure>
  </spaces>
</iq>
```

Response on success:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="configure1"/>
```

The response will either be a simple result IQ without child element, or an error if something failed. The following errors may occur during the configuration:

`item_not_found`	No space exists for the given identifier.
`bad_request`	The space is not defined or the configuration is corrupt.
`not_authorized`	The requesting user is not moderator of the space and therefore not allowed to configure it.

Delete a Space

A `delete` command element triggers the deletion of a space. The space to delete is identified by the `space` attribute.

Request:

```
<iq type="set"
    from="john.doe@mydomain/client"
```

```
    to="spaces.mydomain"
    id="delete1">
  <spaces xmlns="urn:xmpp:spaces">
    <delete space="team#123"/>
  </spaces>
</iq>
```

Response on success:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="delete1"/>
```

As with the configuration of a space, a successful deletion is responded by an empty result. The error conditions are also equal to those of the configuration case.

Handling of Space Channels

The `channels` command element is used to retrieve the channels of a space.

Request:

```
<iq type="get"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="channels1">
  <spaces xmlns="urn:xmpp:spaces">
    <channels space="team#123"/>
  </spaces>
</iq>
```

Response on success:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="channels1">
  <spaces xmlns="urn:xmpp:spaces">
    <channels space="team#123">
      <channel type="pubsub">
        <property key="domain">pubsub.mydomain</property>
        <property key="node">spaces#team#123</property>
      </channel>
      <channel type="muc">
        <property key="jid">team#123@spacemucs.mydomain</property>
      </channel>
    </channels>
  </spaces>
</iq>
```

The channel properties contain the information to address the channel, depending on the channel type. Currently the types `pubsub` and `muc` are supported. They address the publish-subscribe node and, if the space is a team or organizational space, the multi-user chat room JID of the space.

Data Model Support

In organizational spaces, the data exchanged over the publish subscribe channel is restricted to specific data models. To use a organizational space, a whitelist of data models has to be managed. This is done by using the `models` command, which is only available in the context of organizational spaces.

Retrieve Supported Data Models

The list of supported models can be retrieved with the following stanza.

Request:

```
<iq type="get"
    from="john.doe@mydomain/immAndroid"
    to="spaces.mydomain"
    id="models1">
  <spaces xmlns="urn:xmpp:spaces">
    <models space="orga#271"/>
  </spaces>
</iq>
```

Response on success:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/immAndroid"
    id="models1">
  <spaces xmlns="urn:xmpp:spaces">
    <models space="orga#271">
      <model namespace="mirror:application:integratedmoodmap:mood"
schemaLocation="http://data.mirror-demo.eu/application/integratedmoodmap/mo
od-1.0.xsd" />
      <model namespace="mirror:application:integratedmoodmap:topic"
schemaLocation="http://data.mirror-demo.eu/application/integratedmoodmap/to
pic-1.0.xsd" />
    </models>
  </spaces>
</iq>
```

Each entry of the list identifies one data model supported by the organizational space. An entry is a `<model/>` element with two mandatory attributes:

- The `namespace` attribute specifies to the namespace of the supported data model.

- The `schemaLocation` attribute specifies the URL to the XML schema definition of the data model. This schema is used by the spaces service to validate data objects.

The following errors may occur when a `models` command is sent:

`item_not_found`	No space exists for the given identifier.
`not_allowed`	The space contains no model list, i.e., is no organizational space.

Set Supported Data Models

To set the supported data models, an SET-IQ can be used.

Request:

```
<iq type="set"
    from="john.doe@mydomain/client"
    to="spaces.mydomain"
    id="models2">
  <spaces xmlns="urn:xmpp:spaces">
    <models space="orga#271">
      <model namespace="mirror:application:integratedmoodmap:mood"
schemaLocation="http://data.mirror-demo.eu/application/integratedmoodmap/mo
od-1.2.xsd" />
      <model namespace="mirror:application:integratedmoodmap:topic"
schemaLocation="http://data.mirror-demo.eu/application/integratedmoodmap/to
pic-1.0.xsd" />
    </models>
  </spaces>
</iq>
```

Response on success:

```
<iq type="result"
    from="spaces.mydomain"
    to="john.doe@mydomain/client"
    id="models2"/>
```

The syntax from SET command equals to the response from the GET command. Like the space configuration, the provided list of data models given with the request replaces the existing list. The data model list may only be set by space moderators.

The following errors may occur when a `models` command is sent:

`item_not_found`	No space exists for the given identifier.
`not_allowed`	The space contains no model list, i.e., is no organizational space.

`not_authorized`	The requesting user is not moderator of the space and therefore not allowed to set the list of supported data models.