

# The OA4MP Detached/Independent Service

## OA4MP as a detached/independent service

### Overview

The authentication server (AS) may be replaced in OA4MP by any application. This api allows your service to use OA4MP to start and manage the code flow. There are 4 calls that respectively start a flow, then notify OA4MP either if the user has finished authenticating or has cancelled the flow. The service that provides these calls is the **diService** for detached/independent service (your service is independent of OA4MP and detached in the sense that it can reside anywhere).

The scenario is that your AS is at the **/authorize** endpoint and manages all authentication. It simply informs OA4MP about the progress.

### Security and access

All access will be via Tomcat. Since this in effect would make everything world readable, so some form of restriction to access, be it localhost access only or credentials should be used. The default path to the DI servlet will be **http://localhost:8080/oauth2/diService**.

### Setting up OA4MP

#### Setting up Tomcat

In the standard OA4MP distribution, there is a DD (deployment descriptor) **\$OA4MP\_SERVER/etc/WEB-INF/di-service-web.xml** which should be used. After deploying/updating the war, copy this into

**\$CATALINA\_HOME/webapps/oauth2/WEB-INF/web.xml**

and restart. Note that you must do this whenever you upgrade OA4MP.

### The API itself

All access will be via HTTP Get. Technically we should only allow HTTP Post on requests which may result in changes to the server (e.g. a creating a user) and HTTP Get for information requests, however, this would make it awkward to use. The aim is to make it simple enough that clients can hand-code the request and parse the result without having to resort to complex machinery to do so. All requests are via parameters, all responses are JSON.

## Securing the service.

This system presupposes a trust relation and can allow the creation of flows. As such, if it is simply public facing, a denial of service attack would be easy. There 3 main ways to use this service securely.

### 1. Distinguished users.

In this case, the servers the configuration **diService** element explicitly lists users and credentials. Every request must have an entry. This is of use if the AS services all calls and there is no need for more than just authentication, usually by a single, dedicated user. The parameters are **oa4mp:di:user** for the user name and **oa4mp:di:password** for the password.

### 2. Restrict endpoint access

In this case, the endpoint is secured and only accepts requests from specific IP addresses. This gives a slight performance boost and if the AS is running on a dedicated server is a good option. No users are needed and every request from the trusted IP is honored.

### 3. Per Administrative Client

If an admin client has been so flagged, it makes the request using its own (usually RFC 7523) credentials. This permits fine grained control and excellent security if running the service has to be public.

## API Basic Operation

### Making a request

Every request will go to the same address relative endpoint, usually **diService**, e.g.

**`http://your.server.edu:9443/oauth2/diService.`**

Requests are standard HTTP GET with key=value pairs. One of these is required, "action=XXXX", where XXXX determines what is to be done by the server. The remaining key/value pairs are parameters for the call. All arguments in a post or get will be URL encoded but key/value pairs may be in any order. Required arguments must be present or an error will be returned. Generally repeated arguments will cause a duplicate argument message to be generated.

### Format of a response from the server

The general serialized form for an object (which is always the body of the response and is urlencoded) is a JSON object of the form:

```
{
  "status" : XXX,
  "key1" : "value1",
  "key2" : "value2", ...
}
```

Each section below detailing an API call will list what are and are not acceptable values. The status

is always present. The next section details the possible values.

## Status Codes and Error Conditions

It may occur that there are errors during the processing of a request, e.g. if the parameters are incorrect. Rather than confuse server errors with data errors – the bane of RESTful APIs, which this is not, we implement the following policy: Only actual errors with the servlet itself will result in an HTTP status code different than 200. E.g. a 404 Not Found error can only have come from the web server itself and means there is a problem with the request rather than meaning, say, that a user was not found. The only required value of each response is the status. This will be recorded in the body of the response according to the following table. All operations can return a duplicate parameter exception. It is not the task of this servlet to disambiguate or merge conflicting requests. Duplicate arguments are in general not allowed for any argument, including optional ones. All operations can return a missing parameter error for a required parameter.

Note: There is an appendix at the end of this document with various tables sorting these for quick reference.

| Value                     | Decimal | Hex     | Comment  |
|---------------------------|---------|---------|--|
| OK                        | 0       | 0x0     | Normal return  |
| ActionNotFound            | 1       | 0x1     | No such action is supported by this service. Normally this indicates that the action value was mis-typed.    |
| TransactionNotFound       | 1048485 | 0xFFFA5 | No such transaction for the given code or user code was found  |
| DuplicateParameterFound   | 1048561 | 0xFFFF1 | A duplicate argument was supplied.   |
| InternalError             | 1048563 | 0xFFFF3 | Some error internal to the server occurred during processing. Consult the server logs.                       |
| MalformedInputError       | 1048567 | 0xFFFF7 | An input was of the incorrect format. E.g. an illegal uri or a string that cannot be parsed into an integer. |
| MissingParameterError     | 1048569 | 0xFFFF9 | A required argument was not found.   |
| Transaction not found     | 65537   | 10001   | No transaction with the given identifier could be found  |
| Expired token             | 65539   | 10003   | The token for this request has expired.  |
| Create transaction failed | 65541   | 10005   | General exception when a transaction cannot be created   |
| Unknown callback          | 65543   | 10007   | The supplied callback id not in the list of registered callbacks   |
| Client id missing         | 65545   | 10009   | No client identifier has been supplied with this request   |
| No registered callbacks   | 65547   | 1000A   | The client has no callbacks registered.  |

|                   |       |       |  |
|-------------------|-------|-------|--|
|                   |       |       | (Normally this implies an incomplete registration)             |
| Unknown client    | 65549 | 1000C | The identifier does not match any client                       |
| Unapproved client | 65551 | 1000E | This client has been registered but has not yet been approved. |

## Example of a typical error response

```
{
  "description" : "The given redirect_uri is not valid for this client.",
  "error" : "create_transaction_failed",
  "status" : 65541
}
```

# The API Calls

## approveUserCode

**What's it do? *Device Code Flow*.** It will approve the code (allowing the user to get a token). It may also invalidate a user code, meaning the code is flagged as unusable and any attempts to use it thereafter will raise an exception. There is no undo for invalidating a user code. Normally you check the user code (see **checkUserCode**) before approving it.

## Request key/value pairs

| Key       | Value                                  | Comment  |
|-----------|--|--|
| action    | approveUserCode                        | Required   |
| approved  | 0 invalidate<br>1 approve (default)    | Optional. If not present, approve the code, if 0, cancel the flow. |
| auth_time | The time of authorization for the user | Time is in seconds   |
| user_code | The user code generated by the system  | Required.  |
| username  | The name of the user used.             |  |

## Response key/value pairs

| Key    | Value   | Comment |
|--------|---|---------|
| status | Ok<br>missing parameter<br>transaction not found<br>expired user code |         |

|           |                            |   |
|-----------|----------------------------|---|
| client_id | The client id              |   |
| code      | The id for the transaction | This is base 32 encoded                         |
| user_code | The user_code passed in    | This helps the requester identify this response |

## Example continuation, approving the user.

Picking up where **checkUserCode** left off, we assume the user has finished authenticating and the last step for you is to tell OA4MP this:

```
https://localhost:9443/oauth2/diService
?action=approveUserCode
&user_code=684B+7344+XE36
&username=bob@physics.bgsu.edu
&auth_time=1756766314
```

which in turn has a response of

```
{
  "status" : 0,
  "user_code" : "684B+7344+XE36"
  "client_id" : "oa4mp:/client_id/cern/87547887",
  "code" : "J5ETCRRZPFUEEVLPI5FX04KKIY3EQ3RQNFIQ"
}
```

and at this point, the user can simply get a token from OA4MP directly.

See also: **checkUserCode**

Note that if you cancel or invalidate a flow it cannot be restarted and the user must restart.

Note that the client ID is returned so your service can more easily display information (if desired) to the user.

## checkUserCode(user\_code)

**What's it do:** *Device Code Flow*. This will take the user\_code and verify that it is currently active. This is informational so you can manage logon attempts by the user.

### Request key/value pairs.

| Key       | Value                         | Comment  |
|-----------|-------------------------------|----------|
| action    | checkUserCode                 | Required |
| user_code | The user code issued by OA4MP | Required |

### Response key/value pairs.

| Key    | Value | Comment                 |
|--------|-------|-------------------------|
| status | Ok    | The service unavailable |

|           |  |  |
|-----------|--|--|
|           | missing parameter<br>service unavailable<br>transaction not found<br>expired token | response is if the service does not have the device flow enabled.<br>Expired token refers to the auth grant for the transaction. |
| client_id | The client id  |  |
| code      | The id for the transaction   | This is base 32 encoded. Do not alter, just pass it around as needed.  |
| scope     | Scopes, if any, in the initial request   | This is a blank delimited list   |
| user_code | The user_code passed in  | This helps the requester identify this response  |

## Example. Using the device code flow

### Prerequisites

In this example, your service uses a reverse proxy lookup, such as Apache. The entire service has user-facing address at <https://oa4mp.physics.bgsu.edu> and OA4MP is deployed on localhost:9443. Access to the DI Service is restricted to the IP address of your server only and no other authentication is needed.

| External address  | Internal address  | Description   |
|---|---|---|
| <a href="https://hadron.physics.bgsu.edu">https://hadron.physics.bgsu.edu</a>   |   | The general, public address for your service                                  |
| <a href="https://hadron.physics.bgsu.edu/authorize">https://hadron.physics.bgsu.edu/authorize</a>                     | --  | The AS written in e.g. PHP or python. This handles device flow authentication |
| <a href="https://hadron.physics.bgsu.edu/token">https://hadron.physics.bgsu.edu/token</a>                             | <a href="https://localhost:9443/oauth2/token">https://localhost:9443/oauth2/token</a>                               | The OA4MP token endpoint  |
| <a href="https://oa4mp.physics.bgsu.edu/device_authorization">https://oa4mp.physics.bgsu.edu/device_authorization</a> | <a href="https://localhost:9443/oauth2/device_authorization">https://localhost:9443/oauth2/device_authorization</a> | The device authorization endpoint   |
| ... other standard endpoint   | ... their OA4MP addresses   |   |

The user starts by making a request to the device authorization, which is forwarded by Apache. The user gets a response with the **user\_code**:

[https://hadron.physics.bgsu.edu/authorize?user\\_code=684B+7344+XE36](https://hadron.physics.bgsu.edu/authorize?user_code=684B+7344+XE36)

This URI is to your service and is configured in the server configuration in the deviceFlowServlet element, a typical example being:

```
<deviceFlowServlet
  verificationURI="https://hadron.physics.bgsu.edu/authorize"
  interval="5"
  codeChars="0123456789ABCDEFX"
  codeLength="12"
```

```

codeSeparator="+"
codePeriodLength="4"
/>

```

The URL to request this is in OA4MP well-known page, but the configured verification URI can be anywhere. Once the user comes to your service, you can read the `user_code` and query OA4MP to get the client ID, scopes etc (linefeeds added for clarity):

```

https://localhost:9443/oauth2/diService
?action=checkUserCode
&user_code=684B+7344+XE36

```

```

{
  "status" : 0,
  "user_code" : "684B+7344+XE36"
  "client_id" : "oa4mp:/client_id/cern/87547887",
  "scope" : ["write:/", "read:/public/fermilab/grant_3456/data"]
  "code" : "J5ETCRRZPFUEEVLP15FX04KKIY3EQ3RQNF1Q"
}

```

In short everything you need to do authentication and put up a consent page. When the user has completed authentication (or has cancelled it), you would need to notify OA4MP.

See also: `approveUserCode`

## finishAuthCodeFlow

**What's it do?** *Authorization Code Flow*. This sets the user's logon information and signals that authorization is finished. It returns the redirect for the user's browser.

### Request key/value pairs

| Key          | Req? | Value                                     | Comment  |
|--------------|------|---|--|
| action       | Y    | finishAuthCodeFlow                        | Required   |
| approved     | N    | 0 invalidate<br>1 approve (default)       | If the user cancels then setting this to 0 will cancel the flow.   |
| auth_time    | N    | The timestamp when the user authenticated | This is in seconds   |
| myproxy_info | N    | The username MyProxy expects              | When getting x509 certs only. This is a very specific use and unless you are <i>sure</i> you need it, ignore this. |
| code         | Y    | The authorization grant                   | This is the unique identifier for the transaction  |
| username     | Y    | Name of the user at authorization         |  |

### Response key/value pairs

| Key    | Value | Comment                         |
|--------|-------|---------------------------------|
| status | Ok,   | Missing argument is if the code |

|              |   |   |
|--------------|---|---|
|              | missing argument,<br>expired token<br>transaction not found<br>QDL error<br>QDL runtime error | is missing  |
| redirect_uri | The full redirect for the user's browser.   | You set this as a redirect in the response to the user's initial request. |

See also: **startAuthCodeFlow**

Note that if you invalidate a flow it cannot be restarted and the user must restart.

## Example. Finishing the flow

This finishes the flow started in the **startAuthCodeFlow** example. The assumption is that the user has successfully authenticated, you have done everything that is needed and now the very last step for you is to tell OA4MP who the user is and get the correct redirect URL so you can redirect the user's browser. You construct the following call (line breaks added for clarity):

```
http://your.server.edu:9443/oauth2/diService
?action=finishAuthCodeFlow
&code=JMYVIR3HHFBUMVKFGB3F02RVSKZLX06BZIZ2U6MTUKMZFTG0
&username=bob@bgsu.edu
&auth_time=1756732764
```

Getting a response of

```
{ "status": 0,
  "redirect_uri": "https://service.physics.bgsu.edu?code=
JMYVIR3HHFBUMVKFGB3F02RVSKZLX06BZIZ2U6MTUKMZFTG0&state=
2mcya1WBRuMb3agPpLzF8g96"
}
```

In the response to the client, set the URL as the redirect.

## startAuthCodeFlow

**What's it do: Authorization Code Flow.** This is the initial request that creates the transaction. This returns the code (the unique identifier for this flow) and other information.

**Nota Bene: The request to your service is the standard OAuth request and may have any number of parameters based on what the user needs. This table contains the additional parameters to use the API. Add these to the request, but pass along everything else.**

You will still get errors for the OAuth flow if there are any. E.g. the client is configured to require certain scopes and the request is missing these. You will need the code later to finish starting the authorization code flow. Generally, you get the request from a client, change the address to OA4MP and add the action plus any credentials.



Request key/value pairs.

| Key               | Req? | Value   | Comment  |
|-------------------|------|---|----------|
| action            | Y    | startAuthCodeFlow   | Required |
| oa4mp:di:user     | N    | Only needed if the system has users configured for this service |          |
| oa4mp:di:password | N    | “ “ “   |          |

Response key/value pairs.

| Key    | Value  | Comment  |
|--------|--|--|
| status | ok<br>missing argument<br>missing client id<br>no scopes<br>malformed scope<br>malformed input<br>unknown client<br>unapproved client<br>create transaction failed<br>internal error |  |
| code   | The authorization grant  | This is base 32 encoded. Do not alter this, just pass it along as needed.                                    |
| scope  | JSON array   | This is the set of scopes that the client is actually allowed. It may not be the same as what was passed in. |
| state  | Echos back passed in state   | This is so clients can set a value to track transactions   |

See also: **finishAuthCodeFlow**

Example. Servicing a request.

Your service resides at

<http://your.server.edu:9443/oauth2/authorize>

And it gets the following request for OA4MP (URL decoded, linefeeds added for readability):

<http://your.server.edu:9443/oauth2/authorize>  
?response\_type=code  
&scope=org.cilogon.userinfo openid profile email  
&state=2mcyalWBRuMb3agPpLzF8g96  
&...

Before starting to authenticate the user, your service in turn rewrites this as

[http://your.server.edu:9443/oauth2/diService](http://your.server.edu:9443/oauth2/diService?response_type=code&scope=org.cilogon.userinfo+openid+profile+email&state=2mcyaLWBRuMb3agPpLzF8g96&...&action=startAuthCodeFlow)  
?response\_type=code  
&scope=org.cilogon.userinfo openid profile email  
&state=2mcyaLWBRuMb3agPpLzF8g96  
&...  
&action=startAuthCodeFlow

(possibly adding DI username and password if needed) and calls the DI service. Your request gets a response of

```
{
  "status" : 0,
  "code" : "JMYVIR3HHFBUMVKFGB3F02RVSKZLX06BZIZ2U6MTUKMZFTG0",
  "state" : "2mcyaLWBRuMb3agPpLzF8g96"
  "scope" : ["org.cilogon.userinfo","openid","profile","email"]
}
```

This means that the flow transaction has been created in OA4MP. All the information needed to put up the consent page for your AS is returned. Later, once the user has successfully authenticated, you would use the code in the **finishAuthCodeFlow** call.

Note that the initial request may be very long. The contract is to send it *in toto* to the DI Service and let OA4MP decide what to do with it. Your service just adds parameters and changes the address.

## Appendix

The basic philosophy is that even number indicate some sort of success + information, odd numbers represent that an error has happened.

### List of success codes

| NAME      | Hex | Decimal |
|-----------|-----|---------|
| STATUS_OK | 0x0 | 0       |

### Alphabetical table of error codes

| NAME                             | Hex      | Decimal |
|----------------------------------|----------|---------|
| STATUS_ACTION_NOT_FOUND          | 0x1      | 1       |
| STATUS_CLIENT_NOT_FOUND          | 0xFFFFF  | 1048575 |
| STATUS_CREATE_TRANSACTION_FAILED | 0x10005  | 65541   |
| STATUS_DUPLICATE_ARGUMENT        | 0xFFFF1  | 1048561 |
| STATUS_EXPIRED_TOKEN             | 0x10003  | 65539   |
| STATUS_INTERNAL_ERROR            | 0xFFFF3  | 1048563 |
| STATUS_MALFORMED_INPUT           | 0xFFFF7  | 1048567 |
| STATUS_MALFORMED_SCOPE           | 0x10013  | 65555   |
| STATUS_MISSING_ARGUMENT          | 0xFFFF9  | 1048569 |
| STATUS_MISSING_CLIENT_ID         | 0x10009  | 65545   |
| STATUS_NO_SCOPES                 | 0x10011  | 65553   |
| STATUS_PAIRWISE_ID_MISMATCH      | 0x100003 | 1048579 |
| STATUS_QDL_ERROR                 | 0x100007 | 1048583 |
| STATUS_QDL_RUNTIME_ERROR         | 0x100009 | 1048585 |
| STATUS_SERVICE_UNAVAILABLE       | 0x10015  | 65557   |
| STATUS_TRANSACTION_NOT_FOUND     | 0x10001  | 65537   |
| STATUS_TRANSACTION_NOT_FOUND     | 0xFFFA5  | 1048485 |
| STATUS_UNAPPROVED_CLIENT         | 0x1000F  | 65551   |
| STATUS_UNKNOWN_CLIENT            | 0x1000D  | 65549   |

## Error codes sorted by numeric value:

| Hex      | Name                             | Decimal |
|----------|----------------------------------|---------|
| 0x1      | STATUS_ACTION_NOT_FOUND          | 1       |
| 0x10001  | STATUS_TRANSACTION_NOT_FOUND     | 65537   |
| 0x10003  | STATUS_EXPIRED_TOKEN             | 65539   |
| 0x10005  | STATUS_CREATE_TRANSACTION_FAILED | 65541   |
| 0x10009  | STATUS_MISSING_CLIENT_ID         | 65545   |
| 0x1000D  | STATUS_UNKNOWN_CLIENT            | 65549   |
| 0x1000F  | STATUS_UNAPPROVED_CLIENT         | 65551   |
| 0x10011  | STATUS_NO_SCOPES                 | 65553   |
| 0x10013  | STATUS_MALFORMED_SCOPE           | 65555   |
| 0x10015  | STATUS_SERVICE_UNAVAILABLE       | 65557   |
| 0xFFFA5  | STATUS_TRANSACTION_NOT_FOUND     | 1048485 |
| 0xFFFF1  | STATUS_DUPLICATE_ARGUMENT        | 1048561 |
| 0xFFFF3  | STATUS_INTERNAL_ERROR            | 1048563 |
| 0xFFFF7  | STATUS_MALFORMED_INPUT           | 1048567 |
| 0xFFFF9  | STATUS_MISSING_ARGUMENT          | 1048569 |
| 0xFFFFF  | STATUS_CLIENT_NOT_FOUND          | 1048575 |
| 0x100007 | STATUS_QDL_ERROR                 | 1048583 |
| 0x100009 | STATUS_QDL_RUNTIME_ERROR         | 1048585 |