



MyID Enterprise

Version 12.12

MyID Core API

Lutterworth Hall, St Mary's Road, Lutterworth, Leicestershire, LE17 4PS, UK
www.intercede.com | info@intercede.com | [@intercedemyid](https://twitter.com/intercedemyid) | +44 (0)1455 558111

Copyright

© 2001-2024 Intercede Limited. All rights reserved.

Information in this document is subject to change without notice. The software described in this document is furnished exclusively under a restricted license or non-disclosure agreement. Copies of software supplied by Intercede Limited may not be used resold or disclosed to third parties or used for any commercial purpose without written authorization from Intercede Limited and will perpetually remain the property of Intercede Limited. They may not be transferred to any computer without both a service contract for the use of the software on that computer being in existence and written authorization from Intercede Limited.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission of Intercede Limited.

Whilst Intercede Limited has made every effort in the preparation of this manual to ensure the accuracy of the information, the information contained in this manual is delivered without warranty, either express or implied. Intercede Limited will not be held liable for any damages caused, or alleged to be caused, either directly or indirectly by this manual.

Licenses and Trademarks

The Intercede® and MyID® word marks and the MyID® logo are registered trademarks of Intercede in the UK, US and other countries.

Microsoft and Windows are registered trademarks of Microsoft Corporation. Other brands and their products are trademarks or registered trademarks of their respective holders and should be noted as such. All other trademarks acknowledged.

Apache log4net

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

© You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions.

Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. ---

Conventions used in this document

- Lists:
 - Numbered lists are used to show the steps involved in completing a task when the order is important.
 - Bulleted lists are used when the order is unimportant or to show alternatives.
- **Bold** is used for menu items and for labels.

For example:

 - Record a valid email address in '**From**' email address.
 - Select **Save** from the **File** menu.
- *Italic* is used for emphasis:

For example:

 - Copy the file *before* starting the installation.
 - Do *not* remove the files before you have backed them up.
- ***Bold and italic*** hyperlinks are used to identify the titles of other documents.

For example: "See the ***Release Notes*** for further information."

Unless otherwise explicitly stated, all referenced documentation is available on the product installation media.
- A `fixed width` font is used where the identification of spaces is important, including filenames, example SQL queries and any entries made directly into configuration files or the database.
- **Notes** are used to provide further information, including any prerequisites or configuration additional to the standard specifications.

For example:

Note: This issue only occurs if updating from a previous version.
- Warnings are used to indicate where failure to follow a particular instruction may result in either loss of data or the need to manually configure elements of the system.

For example:

Warning: You must take a backup of your database before making any changes to it.

Contents

MyID Core API	1
Copyright	2
Conventions used in this document	6
Contents	7
1 Introduction	11
2 Configuring access	12
2.1 Accessing the API documentation	13
2.2 Accessing the API features	15
2.2.1 Scope	15
3 Server-to-server authentication	16
3.1 Configuring MyID for server-to-server authentication	16
3.1.1 Allowing the Client Credentials OAuth2 logon mechanism	16
3.1.2 Creating a role for the external system	17
3.1.3 Selecting a group for the user account	17
3.1.4 Creating a user account for the external system	18
3.2 Configuring web.oauth2 for server-to-server authentication	18
3.2.1 Creating a shared secret	19
3.2.2 Configuring the authentication service	21
3.3 Obtaining a server-to-server access token	24
3.3.1 Requesting an access token	24
3.3.2 Providing a client identifier	26
4 End-user authentication	27
4.1 Configuring web.oauth2 for end-user based authentication	27
4.1.1 Configuring the authentication service for PKCE	27
4.1.2 Configuring the authentication service for a client secret	30
4.1.3 Cross-origin resource sharing	34
4.2 Obtaining an end-user based access token using PKCE	37
4.2.1 Generating a PKCE code verifier and code challenge	37
4.2.2 Requesting an authorization code	38
4.2.3 Requesting an access token	39
4.3 Obtaining an end-user based access token using a client secret	40
4.3.1 Requesting an authorization code	40
4.3.2 Requesting an access token	41
4.4 Using refresh tokens	42
4.4.1 Configuring the authentication server	43
4.4.2 Obtaining the authorization code	44
4.4.3 Obtaining the access and refresh tokens	44
4.4.4 Using the refresh token to obtain a new access token	45
4.5 Revoking access tokens	46
5 Calling the API	47
5.1 Calling the API from an external system	47
5.2 Calling the API from the documentation	48
5.2.1 Calling the API from the documentation using server-to-server authentication	48

5.2.2 Calling the API from the documentation using end-user authentication	49
6 Operation extension	52
6.1 Obtaining an operation extension token	52
6.2 Obtaining an extension token for Select Security Device	53
7 Managing directories	55
7.1 Sample PowerShell script for managing directories	55
7.1.1 Setting up the web.oauth2 client for the script	56
7.1.2 Refreshing the authentication server settings	57
7.1.3 Setting up the user account for the script	57
7.1.4 Editing the script to use your own server address	57
7.1.5 Running the script	58
7.2 Troubleshooting managing directories	58
8 Reassigning devices	59
8.1 Examples for reassigning a device	60
8.1.1 cURL	60
8.1.2 Python	61
8.1.3 PowerShell	62
8.2 Troubleshooting reassigning devices	63
9 Custom configuration files	64
9.1 Custom client configuration files	65
9.2 Custom scope configuration files	66
9.3 Custom API resource configuration files	67
9.4 Custom identity resource configuration files	68
10 Example – server-to-server	69
10.1 Requirements	70
10.2 Creating a user account	71
10.3 Authenticating to the MyID Core API	72
10.3.1 Generating a client secret	72
10.3.2 Adding the API client	73
10.3.3 Refreshing the authentication server settings	74
10.3.4 Encrypting the client secret	75
10.4 Creating the settings file	76
10.5 Getting an access token	77
10.5.1 Creating a function to obtain an access token	77
10.5.2 Creating the request script	78
10.5.3 Decrypting the secret	78
10.5.4 Adding the decryption call to the request script	79
10.5.5 Obtaining an access token	79
10.5.6 Troubleshooting	80
10.6 Calling the API	81
10.6.1 Setting up the imports and variables	81
10.6.2 Obtaining the access token	81
10.6.3 Calling the API	82
10.6.4 Troubleshooting	83
10.7 Getting a list of group IDs	84

10.7.1 Getting started	86
10.7.2 Processing the results	87
10.7.3 Troubleshooting	87
10.8 Adding a person	88
10.8.1 Setting up the import data	88
10.8.2 Creating your add person script	89
10.8.3 Loading the person's data	89
10.8.4 Calling the API to add a person	90
10.8.5 Checking the response	90
10.8.6 Troubleshooting	91
10.9 Updating a person	93
10.9.1 Creating your update person script	93
10.9.2 Setting up the update data	93
10.9.3 Finding the ID of the person	94
10.9.4 Calling the API to update a person	95
10.9.5 Troubleshooting	96
10.10 Uploading a picture	97
10.10.1 Providing an image	97
10.10.2 Creating your upload image script	97
10.10.3 Loading the image from a file	98
10.10.4 Setting up the update data	98
10.10.5 Finding the ID of the person	98
10.10.6 Calling the API to upload a picture	99
10.10.7 Troubleshooting	100
10.11 Getting credential profiles	101
10.11.1 Creating your update person script	101
10.11.2 Finding the ID of the person	101
10.11.3 Calling the API to get a list of credential profiles	102
10.11.4 Troubleshooting	103
10.12 Requesting a device	104
10.12.1 Creating your request device script	104
10.12.2 Finding the ID of the person	104
10.12.3 Setting the ID of the credential profile	105
10.12.4 Calling the API to request a device	106
10.12.5 Troubleshooting	107
10.13 Checking the status of a request	108
10.13.1 Creating your request status script	108
10.13.2 Setting the ID of the request	108
10.13.3 Calling the API to get the request status	109
10.13.4 Creating your notification routines	109
10.13.5 Checking the status of the request	110
10.13.6 Troubleshooting	110
10.14 Getting multiple pages of reports	111
10.14.1 Creating your report paging script	111
10.14.2 Creating a function to return a specific page of a report	112

10.14.3 Setting up the variables	112
10.14.4 Calling the report page function	112
10.14.5 Providing the results	113
10.14.6 Troubleshooting	113
10.15 Working with JWTs	114
10.15.1 Creating your JWT script	114
10.15.2 Decoding the JWT	115
10.15.3 Verifying the signature	116
10.15.4 Troubleshooting	116
11 Example – user authentication	117
11.1 Requirements	117
11.2 Authenticating to the MyID Core API	118
11.2.1 Adding the API client	118
11.2.2 Configuring the MyID Core API for CORS	122
11.2.3 Refreshing the web server settings	123
11.3 Obtaining an authorization code	124
11.4 Creating a customizable form	128
11.5 Obtaining an access token	132
11.6 Generating a random code verifier and challenge	135
11.7 Calling the API	138
11.8 Displaying a list of people	139
11.9 Displaying the details for a person	141
11.9.1 Exploring the links	144
11.10 Example code listing	145
12 Troubleshooting	149

1 Introduction

The MyID® Core API gives you access to the features used in the MyID Operator Client using a REST-based API, allowing you to integrate into other business systems that provide information to MyID or to trigger credential lifecycle events.

The API is secure by default, requiring authentication of the calling system in order to use the API and restricting access to available features and data using MyID role-based access and scope control.

You can use the API for such actions as:

- Searching and retrieving information about a person, device or request.
- Adding or updating a person's information in MyID.
- Managing the lifecycle of people, devices, and requests.

Comprehensive API documentation is provided, including schema information to simplify integration development.

2 Configuring access

This section provides details of configuring access to the API:

- For details of providing access to the built-in API documentation, see section [2.1](#), *Accessing the API documentation*.
- For details of configuring access to the features of the API, see section [2.2](#), *Accessing the API features*.

2.1 Accessing the API documentation

The API documentation is provided on the API server at the following address:

`https://<myserver>/rest.core/swagger/index.html`

where `<myserver>` is the name of the server hosting the web service.

If necessary, you can configure the API documentation to prevent access. You can also to configure the API documentation to provide details of error codes, and to allow you to use the documentation as a test harness, subject to the appropriate authentication.

To configure access the API documentation:

1. In a text editor, open the `appsettings.Production.json` file for the web service.

By default, this is:

`C:\Program Files\Intercede\MyID\rest.core\appsettings.Production.json`

This file is the override configuration file for the `appsettings.json` file for the web service. If this file does not already exist, you must create it in the same folder as the `appsettings.json` file, and include the following:

```
{
  "MyID": {
    "SwaggerApiDocumentation": {
      "GenerateDocumentation": true,
      "DocumentErrorCodes": false,
      "DocumentMIReports": true,
      "AllowTestFromDocumentation": false,
      "ShowPermissions": false,
      "OrderAlphabetically": true
    }
  }
}
```

If the `appsettings.Production.json` file already exists, add the above `SwaggerApiDocumentation` section to the file.

2. Edit the following values:

- `GenerateDocumentation` – set to `false` to prevent access to the documentation.
- `DocumentErrorCodes` – set to `true` to include details of the web service error messages associated with HTTP status codes.

Note: If you are consuming the Swagger output using an external tool, you may experience compatibility issues if you set this value to `true`; set `DocumentErrorCodes` to `false` to prevent this.

- `DocumentMIReports` – set to `true` to include details of Management Information reports.
- `AllowTestFromDocumentation` – set to `true` to allow you to use the documentation as a test harness, subject to the appropriate authentication. When you enable this option, the API documentation provides authentication instructions at the top of the page; see also section [5.2, Calling the API from the documentation](#).

- `ShowPermissions` – set to `true` to display the permissions required for each API call. See section [2.2, Accessing the API features](#) for more information.
 - `OrderAlphabetically` – set to `true` to order the endpoints alphabetically within each category in the documentation. Set to `false` to return to unordered endpoints.
3. Save the `appsettings.Production.json` file.
 4. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.rest.core.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

5. Open a web browser, and navigate to the following URL:

`https://<myserver>/rest.core/swagger/index.html`

where `<myserver>` is the name of the server hosting the web service.

2.2 Accessing the API features

Access to features of the MyID Core API is controlled using MyID roles.

For example, the MyID Operator Client feature that allows you to view a person's images (View Person's Images) is enabled if the operator has a role with one of the following permissions:

- Add Person
- Edit Person
- Request Card
- Request Replacement Card
- View Person

If the operator account has access to *any* of these permissions, it can use the corresponding API call:

- `GET /api/People/{id}/images/{imageField}`

For information on setting role permissions, see the *Roles* section in the [Administration Guide](#).

To determine what permissions are required for an API call, set the `ShowPermissions` option to `true` in the `rest.core` configuration file; see section 2.1, [Accessing the API documentation](#) for details.

This adds a section to the API documentation that lists the permissions for each API call, and which roles currently have access:

GET /api/People/{id}/images/{imageField} Retrieve an image that is associated with the Person

Retrieve an image that is associated with the Person.
The image will be returned as a binary object with the correct MIME type set.

Operation ID	Granted By	Available To
100109	Add Person Edit Person View Person Request Card Request Replacement Card Approve Person Unapprove Person	Operator Personnel SecurityGroupC Startup User System Help Desk Manager PasswordUser Security Chief SecurityGroupB Cardholder SecurityGroupA

If there are no workflows listed in the **Granted By** list, this means you cannot enable access to the endpoint; this endpoint is reserved for future use.

Note: You may find some API features that do not have role-based restrictions placed on them, and the **Operation ID | Granted By | Available To** table is not displayed; however, the object of the operations will always respect the scope of the operator user.

2.2.1 Scope

The MyID Core API respects the scope of the operator account used to access the API. For example, if you are using an operator account in the Finance department that has a role with a scope of Department, that account can view and access only the people (and their devices, requests, and so on) who are in the Finance department.

For information on setting roles and scope permissions, see the *Scope and security* section in the [Administration Guide](#).

3 Server-to-server authentication

This section provides information on setting up server-to-server authentication for the MyID Core API. This allows you to call the MyID Core API from an external system.

To set up server-to-server authentication, you must carry out the following:

- Configure MyID with a user account that has the appropriate permissions to access the API.

See section [3.1, Configuring MyID for server-to-server authentication](#).

- Configure the web.oauth2 web service with the ID of your external system and a shared secret.

See section [3.2, Configuring web.oauth2 for server-to-server authentication](#).

- Obtain an access token for your system to use.

See section [3.3, Obtaining a server-to-server access token](#).

For more information, see the *Client Credentials* section of the *OAuth 2.0 Authorization Framework*:

tools.ietf.org/html/rfc6749#section-1.3.4

3.1 Configuring MyID for server-to-server authentication

The MyID Core API uses a user account to log on to the MyID system. This allows you to configure access to particular MyID features and groups using the standard roles, groups, and scope feature of MyID. All actions carried out through the API are audited under this user account, and if necessary you can disable the account to prevent access to the API.

3.1.1 Allowing the Client Credentials OAuth2 logon mechanism

To allow access to MyID through server-to-server authentication, you must enable the Client Credentials OAuth2 logon mechanism.

1. In MyID Desktop, from the **Configuration** category, select **Security Settings**.
2. On the **Logon Mechanisms** tab, set the following option:
 - **Client Credentials OAuth2 Logon** – set to Yes to enable server-to-server authentication, or No to disable it.
3. Click **Save changes**.

3.1.2 Creating a role for the external system

You are recommended to create a new role to be used for controlling access to MyID from the external system, rather than using an existing role. This allows you to maintain clear control over the MyID features the external system can access.

To create the role:

1. In MyID Desktop, from the **Configuration** category, select **Edit Roles**.
2. Click **Add**.
3. Give the role a name; for example, `External API`.
4. From the **Derived from** drop-down list, select **Allow None**.
5. Click **Add**.
6. Select the options that relate to the API features you want to be able to access through the API.

See section [2.2, Accessing the API features](#) for a list of which options map to the API end points.

You are strongly recommended to select only those options that your external system will need to use.

7. Click **Logon Methods**.
8. For the role you created, select the **Client Credentials OAuth2** logon mechanism, then click **OK**.
9. Click **Save Changes**.

For more information about using the **Edit Roles** workflow, see the *Roles* section in the [Administration Guide](#).

3.1.3 Selecting a group for the user account

Before you create the user account, you must consider into which group you want to put the account. The group you select affects the scope of the user.

- If you want to restrict the access of the API to a particular group of users in MyID, put the API user into the same group, then select a scope of Department or Division when you specify the role for the user account.
- If you do not want to restrict the access of the API, you are recommended to create a separate group for the API user, then select a scope of All when you specify the role. Use the **Add Group** workflow in MyID Desktop to create the group; you can restrict this group to the API role only, and assign this as the default role with a scope of All.

See the *Adding a group* section in the [Operator's Guide](#) for details of adding groups, and the *Default roles* section in the [Administration Guide](#) for details of setting default roles.

3.1.4 Creating a user account for the external system

Once you have created the role and decided which group to use, you can create the user account that the API will use to access MyID.

To add the user account:

1. In the MyID Operator Client, select the **People** category.
2. Click **Add**.
3. Provide a **First Name** and **Last Name**; for example, `External API`.
4. Provide a **Logon** name; for example, `api.external`.
5. Select a **Group** for the user account.
See section [3.1.3, *Selecting a group for the user account*](#) above for considerations.
6. Select the **Roles** for the user account.
Select the role you created for use by the external system. Set the appropriate scope; for example, **All** to allow the API to access data related to any user account in the system, or **Division** to restrict access data related to accounts in the same group as the API user, along with any subgroups.
7. Click **Save**.

Make sure you take a note of the logon name for the user; you need this for configuring the web.oauth2 web service.

3.2 Configuring web.oauth2 for server-to-server authentication

The MyID authentication web service is called web.oauth2; you must configure this web service to allow access to the API. For server-to-server authentication, you do this using a client credential grant, which you request using a shared secret.

Note: Before you begin, you must decide on a client ID for your external system; for example, `myid.mysystem`. This represents your back-end system that intends to make calls to the MyID Core API.

3.2.1 Creating a shared secret

Important: You must keep the shared secret safe. This information can be used to grant authorization to the API, so must be an unguessable value; for this reason, you are recommended to generate a GUID for the shared secret.

To create a shared secret:

1. Generate a GUID to use as the secret.

For example:

```
82564d6e-c4a6-4f64-a6d4-cac43781c67c
```

2. Create a hash of this GUID using SHA-256, and convert it to Base64.

For example:

```
kv31VP5z/oKS0QMMaIfZ2UrhmqOdgAPpXV/vaF1cymk=
```

You need this value for the web.oauth2 server. The server does not store the secret, only the hash.

Important: Do not use this example secret in your own system.

3. Combine your client ID, a colon, and the GUID secret:

For example:

```
myid.mysystem:82564d6e-c4a6-4f64-a6d4-cac43781c67c
```

4. Convert this string to Base64.

For example:

```
bXlpZC5teXN5c3RlbTo4MjU2NGQ2ZS1jNGE2LTRmNjQtYTZkNC1jYWM0Mzc4MWM2N2M=
```

This is the value you will send in the Authorization header for HTTP basic authentication when requesting the access token; alternatively, you can pass the client ID and the shared secret separately in the body as `client_id` and `client_secret` parameters.

3.2.1.1 Example

The following PowerShell example script shows the process for generating a shared secret and creating the hash and Base64 versions you need to configure the web.oauth2 server and request an access token.

```
# Set the client ID of your calling system
$client_id = "myid.mysystem"

# Generate a new GUID to use as the shared secret
$secret = (New-Guid).ToString()

# Hash the new GUID using SHA-256
$hasher = [System.Security.Cryptography.HashAlgorithm]::Create("sha256")
$hashOfSecret = $hasher.ComputeHash([System.Text.Encoding]::UTF8.GetBytes($secret))

# Convert the hashed secret to Base64
$clientSecret = [Convert]::ToBase64String($hashOfSecret)

# Combine the client ID and secret into a single Base64 authorization token
$both = "$client_id`:$secret"
$bytes = [System.Text.Encoding]::UTF8.GetBytes($both)
$combined = [Convert]::ToBase64String($bytes)

# Output the results
Write-Output ("`r`nThe shared secret is: `r`n`r`n$secret")
Write-Output ("`r`nAnd the hash of the shared secret in base64 is:`r`n`r`n$clientSecret" )
Write-Output ("`r`nStore this value in the ClientSecrets of the web.oauth2 appsettings
file.")
Write-Output ("`r`nThe combined string of the client ID and the shared secret
is:`r`n`r`n$both")
Write-Output ("`r`nAnd in base64 this is: `r`n`r`n$combined")
Write-Output ("`r`nUse this value to request an access token from the web service.")

# Wait for a keypress
Write-Host "`r`nPress any key to continue...`r`n" -ForegroundColor Yellow
[void][System.Console]::ReadKey($true)
```

Example output:

```
PS C:\Intercede> .\secret.ps1

The shared secret is:

82564d6e-c4a6-4f64-a6d4-cac43781c67c

And the hash of the shared secret in base64 is:

kv31VP5z/oKS0QMMaIfZ2Urhmq0dGAppXV/vaF1cymk=

Store this value in the ClientSecrets of the web.oauth2 appsettings file.

The combined string of the client ID and the shared secret is:

myid.mysystem:82564d6e-c4a6-4f64-a6d4-cac43781c67c

And in base64 this is:

bXlpZC5teXN5c3R1bTo4MjU2NGQ2ZS1jNGE2LTRmNjQtYTZkNC1jYWw0Mzc4MWM2N2M=

Use this value to request an access token from the web service.

Press any key to continue...
```

3.2.2 Configuring the authentication service

Once you have created a Base64 version of the hash of the shared secret, you can configure the web.oauth2 server.

First, create a custom client configuration file.

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings.json` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any .json file in the CustomClients folder.
- appsettings.Production.json
- appsettings.json

Note: The appsettings.Production.json file overrides the appsettings.json file *not* by using the client ID, but by the index of the entry in the Clients array. For this reason, you are recommended *not* to use the appsettings.Production.json file to provide the details of custom clients, but to use separate files in the CustomClients folder instead.

For more information about custom configuration files, see section 9, [Custom configuration files](#).

Now that you have a custom client configuration file, you can add your client details:

1. Edit the new file to include the following:

```
{
  "ClientId": "<my client ID>",
  "ClientName": "<my client name>",
  "AccessTokenLifetime": "<time>",
  "AllowedGrantTypes": [
    "client_credentials"
  ],
  "ClientSecrets": [
    {
      "Value": "<secret>"
    }
  ],
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "Properties": {
    "MyIDLogonName": "<my user account>"
  }
}
```

where:

- <my client ID> – the client ID you decided on; for example:
myid.mysystem
- <my client name> – an easily readable name for your client system; for example:
My External System
- <time> – the time (in seconds) that the client credential is valid. The default is 3600 – 1 hour.
- <secret> – the Base64-encoded SHA-256 hash of the secret you created; for example:
kv31VP5z/oKS0QMMaIfZ2UrhmqOdGAPpXV/vaF1cymk=
- <my user account> – the logon name of the user you created to run the API; for example:
api.external

For example:

```
{
  "ClientId": "myid.mysystem",
  "ClientName": "My External System",
  "AccessTokenLifetime": "3600",
  "AllowedGrantTypes": [
    "client_credentials"
  ],
  "ClientSecrets": [
    {
      "Value": "kv31VP5z/oKS0QMMaIfZ2Urhmq0dgAPpXV/vaF1cymk="
    }
  ],
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "Properties": {
    "MyIDLogonName": "api.external"
  }
}
```

2. Save the configuration file.
3. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

4. Check that the web.oauth2 server is still operational by logging on to the MyID Operator Client.

Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

HTTP Error 500.30 - ANCM In-Process Start Failure

See section [12, Troubleshooting](#) for information on resolving problems that cause HTTP Error 500.30.

3.3 Obtaining a server-to-server access token

Once you have configured MyID to allow server-to-server access, set up the user account for the API, configured the shared secret, and set up the web.oauth2 web service to recognize your external system, you can request an access token that you can then use to call the API.

3.3.1 Requesting an access token

Request the access token from the following location:

`https://<myserver>/web.oauth2/connect/token`

POST a request in `application/x-www-form-urlencoded` format.

You must provide the following parameters:

- `grant_type=client_credentials`
- `scope=myid.rest.basic`

You must also provide an `Authorization` header containing "Basic " followed by your client ID and shared secret, combined in a single Base64 string.

For example, if your client ID is:

`myid.mysystem`

and the secret is:

`82564d6e-c4a6-4f64-a6d4-cac43781c67c`

the combination is:

`myid.mysystem:82564d6e-c4a6-4f64-a6d4-cac43781c67c`

and the Base64 string is:

`bXlpZC5teXN5c3RlbTo4MjU2NGQ2ZS1jNGE2LTRmNjQtYTZkNC1jYWM0Mzc4MWM2N2M=`

and the authorization token is:

`Basic bXlpZC5teXN5c3RlbTo4MjU2NGQ2ZS1jNGE2LTRmNjQtYTZkNC1jYWM0Mzc4MWM2N2M=`

Important: Do not use this example secret in your own system.

For example (using cURL):

```
curl -k -i -H "Content-Type: application/x-www-form-urlencoded" -X POST
"https://myserver.example.com/web.oauth2/connect/token" -d "grant_type=client_
credentials&scope=myid.rest.basic" -H "Authorization: Basic
bXlpZC5teXN5c3R1bTo4MjU2NGQ2ZS1jNGE2LTRmNjQtYTZkNC1jYWw0Mzc4MWM2N2M="
```

or using PowerShell:

```
$combined = "bXlpZC5teXN5c3R1bTo4MjU2NGQ2ZS1jNGE2LTRmNjQtYTZkNC1jYWw0Mzc4MWM2N2M="

# Set up the body of the request
$body = @{grant_type='client_credentials'
        scope='myid.rest.basic'
        }
# Set up the header of the request
$header = @{'Content-Type'='application/x-www-form-urlencoded'
            Authorization="Basic $combined"
            }

# Request the access token
Invoke-WebRequest -Method POST -Uri
'https://myserver.example.com/web.oauth2/connect/token' -body $body -Headers $header |
Select-Object -Expand Content

#Wait for a keypress
Write-Host "`r`nPress any key to continue..." -ForegroundColor Yellow

[void][System.Console]::ReadKey($true)
```

An alternative method, passing the `client_id` and `client_secret` in the body rather than in the header:

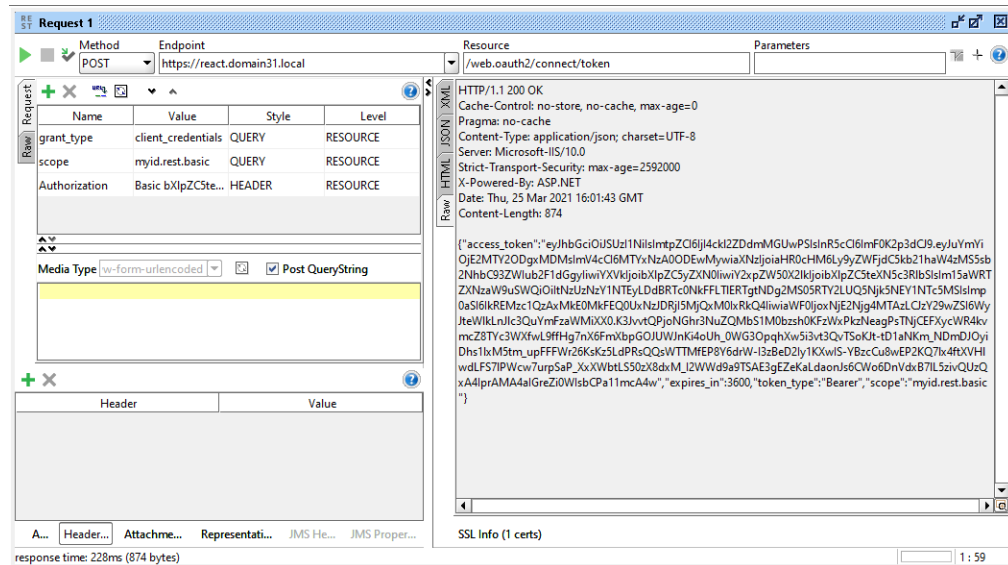
```
# Set up the body of the request
$body = @{grant_type='client_credentials'
        scope='myid.rest.basic'
        client_id='myid.mysystem'
        client_secret='82564d6e-c4a6-4f64-a6d4-cac43781c67c'
        }
# Set up the header of the request
$header = @{'Content-Type'='application/x-www-form-urlencoded'
            }

# Request the access token
Invoke-WebRequest -Method POST -Uri
'https://myserver.example.com/web.oauth2/connect/token' -body $body -Headers $header |
Select-Object -Expand Content

#Wait for a keypress
Write-Host "`r`nPress any key to continue..." -ForegroundColor Yellow

[void][System.Console]::ReadKey($true)
```

You can also use utilities such as SoapUI:



The request returns a block of JSON containing the following:

- `access_token` – your access token.
- `expires_in` – the lifetime in seconds for the token. Once the lifetime has expired, you must request a new access token.
- `token_type` – always `Bearer`.
- `scope` – the scope configured for the client in the web.outh2 web service; usually `myid.rest.basic`.

You can now use this access token to call the API.

See section 5.1, *Calling the API from an external system* for more information on using an access token.

3.3.2 Providing a client identifier

MyID captures the IP address and the client identifier of the PC used to carry out the audited operation, and stores this information in the audit trail; see the *Logging the client IP address and identifier* section in the [Administration Guide](#) for more information.

You can provide a client identifier for your grant request by setting a value for the `CLIENT_IDENTIFIER` header in the request.

Note: You can change the name of the header if required; the name of the header is specified in the `MyID:ClientIdentifierHeader` of the `appSettings.json` file of the web.outh2 web service.

4 End-user authentication

This section provides information on setting up end-user based authentication for the MyID Core API. This allows you to call the MyID Core API using the credentials of a person in the MyID system, using the MyID authentication service to authenticate their credentials, whether security phrases, smart card, FIDO authenticator, or any other authentication method for which MyID is configured.

To set up end-user based authentication, you must carry out the following:

- Configure the web.oauth2 web service with the ID of your client system.
See section [4.1, Configuring web.oauth2 for end-user based authentication](#).
- Authenticate your user account and obtain an access token for your system to use.
See section [4.2, Obtaining an end-user based access token using PKCE](#) or section [4.3, Obtaining an end-user based access token using a client secret](#).
- Optionally, configure your system to allow refresh tokens.
See section [4.4, Using refresh tokens](#).
- Revoke access tokens when they are no longer required.
See section [4.5, Revoking access tokens](#).

For more information on end-user authentication, see the *Authorization Code* section of the *OAuth 2.0 Authorization Framework*:

tools.ietf.org/html/rfc6749#section-1.3.1

4.1 Configuring web.oauth2 for end-user based authentication

The MyID authentication web service is called web.oauth2; you must configure this web service to allow access to the API. For end-user based authentication, you do this by configuring the server for acquisition of an access token, secured either by PKCE or a client secret.

Note: Before you begin, you must decide on a client ID for your external system; for example, `myid.myclient`. This represents your application (for example, website) that intends to make calls to the MyID Core API.

4.1.1 Configuring the authentication service for PKCE

For single-page apps, which run entirely on the client PC, you must secure the request for authentication using PKCE. This ensures that only the caller of the authorization can use the authorization code to request an access token.

First, create a custom client configuration file.

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomClients` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the client ID, but by the index of the entry in the `Clients` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom clients, but to use separate files in the `CustomClients` folder instead.

For more information about custom configuration files, see section 9, [Custom configuration files](#).

Now that you have a custom client configuration file, you can add your client details.

1. Edit the new file to include the following:

```
{
  "ClientId": "<my client ID>",
  "ClientName": "<my client name>",
  "AccessTokenLifetime": <time>,
  "AllowedGrantTypes": [
    "authorization_code"
  ],
  "RequireClientSecret": false,
  "RequirePkce": true,
  "AllowAccessTokensViaBrowser": true,
  "RequireConsent": false,
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "RedirectUris": [
    "<callback URL>"
  ]
}
```

where:

- <my client ID> – the client ID you decided on; for example:
myid.mysystem
- <my client name> – an easily readable name for your client system; for example:
My Client System
- <time> – the time (in seconds) that the client credential is valid. The default is 3600 – 1 hour.
- <callback URL> – the URL of the web page on your system to which the authorization code will be returned.

For example:

```
{
  "ClientId": "myid.myclient",
  "ClientName": "My Client System",
  "AccessTokenLifetime": 3600,
  "AllowedGrantTypes": [
    "authorization_code"
  ],
  "RequireClientSecret": false,
  "RequirePkce": true,
  "AllowAccessTokensViaBrowser": true,
  "RequireConsent": false,
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "RedirectUri": [
    "https://myserver/mysystem/callback.asp"
  ]
}
```

2. Save the configuration file.
3. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

4. Check that the web.oauth2 server is still operational by logging on to the MyID Operator Client.

Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

HTTP Error 500.30 - ANCM In-Process Start Failure

See section [12, Troubleshooting](#) for information on resolving problems that cause HTTP Error 500.30.

You can now obtain an access token; see section [4.2, Obtaining an end-user based access token using PKCE](#).

4.1.2 Configuring the authentication service for a client secret

For stateful websites, where for example the server uses cookies to map stateful sessions between the client and the web server, it is recommended to configure the authentication service to require a client secret; you do not have to use PKCE, but you can use it in addition to the client secret if you want.

Note: The following instructions assume that you are using a client secret without PKCE. If you want to use both a client secret *and* PKCE, you can set both the `RequireClientSecret`

and `RequirePkce` options to `true`, and then combine the requests for an authentication code and access token from section 4.2, *Obtaining an end-user based access token using PKCE* and section 4.3, *Obtaining an end-user based access token using a client secret*.

Before you edit the configuration file, create a client secret; see section 3.2.1, *Creating a shared secret*.

First, create a custom client configuration file.

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomClients` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the client ID, but by the index of the entry in the `Clients` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom clients, but to use separate files in the `CustomClients` folder instead.

For more information about custom configuration files, see section 9, [Custom configuration files](#).

Now that you have a custom client configuration file, you can add your client details:

1. Edit the new file to include the following:

```
{
  "ClientId": "<my client ID>",
  "ClientName": "<my client name>",
  "AccessTokenLifetime": <time>,
  "AllowedGrantTypes": [
    "authorization_code"
  ],
  "RequireClientSecret": true,
  "RequirePkce": false,
  "AllowAccessTokensViaBrowser": true,
  "RequireConsent": false,
  "ClientSecrets": [
    {
      "Value": "<secret>"
    }
  ],
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "RedirectUris": [
    "<callback URL>"
  ]
}
```

where:

- `<my client ID>` – the client ID you decided on; for example:
`myid.mysystem`
- `<my client name>` – an easily readable name for your client system; for example:
`My Client System`
- `<time>` – the time (in seconds) that the client credential is valid. The default is 3600 – 1 hour.
- `<secret>` – the Base64-encoded SHA-256 hash of the secret you created; for example:
`kv31VP5z/oKS0QMMaIfZ2UrhmqOdGAPpXV/vaF1cymk=`
- `<callback URL>` – the URL of the web page on your system to which the authorization code will be returned.

For example:

```
{
  "ClientId": "myid.myclient",
  "ClientName": "My Client System",
  "AccessTokenLifetime": 3600,
  "AllowedGrantTypes": [
    "authorization_code"
  ],
  "RequireClientSecret": true,
  "RequirePkce": false,
  "AllowAccessTokensViaBrowser": true,
  "RequireConsent": false,
  "ClientSecrets": [
    {
      "Value": "kv31VP5z/oKS0QMMaIfZ2Urhmq0dgAPpXV/vaF1cymk="
    }
  ],
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "RedirectUri": [
    "https://myserver/mysystem/callback.asp"
  ]
}
```

2. Save the configuration file.
3. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

4. Check that the web.oauth2 server is still operational by logging on to the MyID Operator Client.

Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

HTTP Error 500.30 - ANCM In-Process Start Failure

See section [12, Troubleshooting](#) for information on resolving problems that cause HTTP Error 500.30.

You can now obtain an access token; see section [4.3, Obtaining an end-user based access token using a client secret](#).

4.1.3 Cross-origin resource sharing

Cross-origin resource sharing (CORS) defines a way for web applications on one domain to interact with resources on other domains; for example, if you create a website on one domain that uses end-user authentication, then using the access token to call the MyID Core API on another domain, you must configure the MyID web.oauth2 and rest.core services to allow CORS.

To allow CORS:

1. In a text editor, open the custom client configuration file you created.

By default, this is:

C:\Program Files\Intercede\MyID\web.oauth2\appsettings.Production.json

2. In the client you set up for your external system, add the following:

```
"AllowedCorsOrigins": [  
  "<external origin>"  
]
```

where:

- <external origin> is the URL from which you are going to call the API. You can add multiple origins if necessary.

Note: Make sure you use an origin, and not an URL, when configuring CORS. For example: `https://myserver/` is an URL, while `https://myserver` is an origin.

For example:

```
{  
  "ClientId": "myid.myclient",  
  "ClientName": "My Client System",  
  "AccessTokenLifetime": 3600,  
  "AllowedGrantTypes": [  
    "authorization_code"  
  ],  
  "RequireClientSecret": true,  
  "RequirePkce": false,  
  "AllowAccessTokensViaBrowser": true,  
  "RequireConsent": false,  
  "ClientSecrets": [  
    {  
      "Value": "kv31VP5z/oKS0QMMaIfZ2Urhmq0dgAPpXV/vaF1cymk="  
    }  
  ],  
  "AllowedScopes": [  
    "myid.rest.basic"  
  ],  
  "RedirectUris": [  
    "https://myserver/mysystem/callback.asp"  
  ],  
  "AllowedCorsOrigins": [  
    "http://myexternalserver"  
  ]  
}
```

3. Save the configuration file and recycle the web.oauth2 app pool.
4. In a text editor, open the `appsettings.Production.json` file for the rest.core service.

By default, this is:

`C:\Program Files\Intercede\MyID\rest.core\appsettings.Production.json`

This file is the override configuration file for the `appsettings.json` file for the web service. If this file does not already exist, you must create it in the same folder as the `appsettings.json` file, and include the following:

```
{
  "MyID": {
    "Cors": {
      "AllowedOrigins": [
        "<external origin>"
      ],
    },
  },
}
```

where:

- `<external origin>` is the URL from which you are going to call the API. You can add multiple origins if necessary.

Note: Make sure you use an origin, and not an URL, when configuring CORS. For example: `https://myserver/` is an URL, while `https://myserver` is an origin.

If the `appsettings.Production.json` file already exists, add the above `Cors:AllowedOrigins` section to the file.

For example:

```
{
  "MyID": {
    "Cors": {
      "AllowedOrigins": [
        "http://myexternalserver"
      ],
    },
  },
}
```

5. Save the configuration file and recycle the rest.core app pool.

4.1.3.1 Troubleshooting CORS

If you see an error similar to the following in your browser console:

```
Access to XMLHttpRequest at 'https://myserver/rest.core/api/People?q=*'  
from origin 'http://myexternalserver:5500' has been blocked by CORS policy:  
No 'Access-Control-Allow-Origin' header is present on the requested  
resource.
```

suggests that the `AllowedOrigins` option in the `rest.core` `appsettings.Production.json` file, or the `AllowedCorsOrigins` option in the `web.oauth2` custom client configuration file, has not been set up correctly.

4.2 Obtaining an end-user based access token using PKCE

Once you have configured the web.oauth2 web service for PKCE, you can request your user-based authentication code, and use that to obtain an access token that you can use to call the API.

For more information on PKCE, including details of requirements for the code verifier and code challenge, see the *Proof Key for Code Exchange by OAuth Public Clients* RFC:

tools.ietf.org/html/rfc7636

4.2.1 Generating a PKCE code verifier and code challenge

The PKCE code verifier and code challenge are used to request the authorization code.

1. Generate a cryptographically-random key.

This is the *code verifier*.

The code verifier must be a high-entropy cryptographic random string using the following characters:

[A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~"

The minimum length is 43 characters, and the maximum length is 128 characters.

2. Generate a SHA-256 hash of this key, then encode it using Base64 URL encoding.

This is the *code challenge*.

Important: Base64 URL encoding is slightly different to standard Base64 encoding.

See the *Protocol* section of the PKCE standard for details of requirements for the code verifier and code challenge:

tools.ietf.org/html/rfc7636#section-4

Example PowerShell script for generating a code challenge from a given code verifier:

```
$code_verifier = 'TiGVEDHIRkdTpif4zLw8v6tcdG2VJXvP4r0fuLhsXIj'

# Hash the code verifier using SHA-256
$hasher = [System.Security.Cryptography.HashAlgorithm]::Create("sha256")
$hashOfSecret = $hasher.ComputeHash([System.Text.Encoding]::UTF8.GetBytes($code_verifier))

# Convert to Base64 URL encoded (slightly different to normal Base64)
$clientSecret = [System.Convert]::ToBase64String($hashOfSecret)
$clientSecret = $clientSecret.Split('=')[0]
$clientSecret = $clientSecret.Replace('+', '-')
$clientSecret = $clientSecret.Replace('/', '_')

# Output the results
Write-Output "`n`nThe code verifier is: `n`n`n$code_verifier"
Write-Output "`n`nAnd code challenge is: `n`n`n$clientSecret" )

# Wait for a keypress
Write-Host "`n`nPress any key to continue...`n`n" -ForegroundColor Yellow
[void][System.Console]::ReadKey($true)
```

4.2.2 Requesting an authorization code

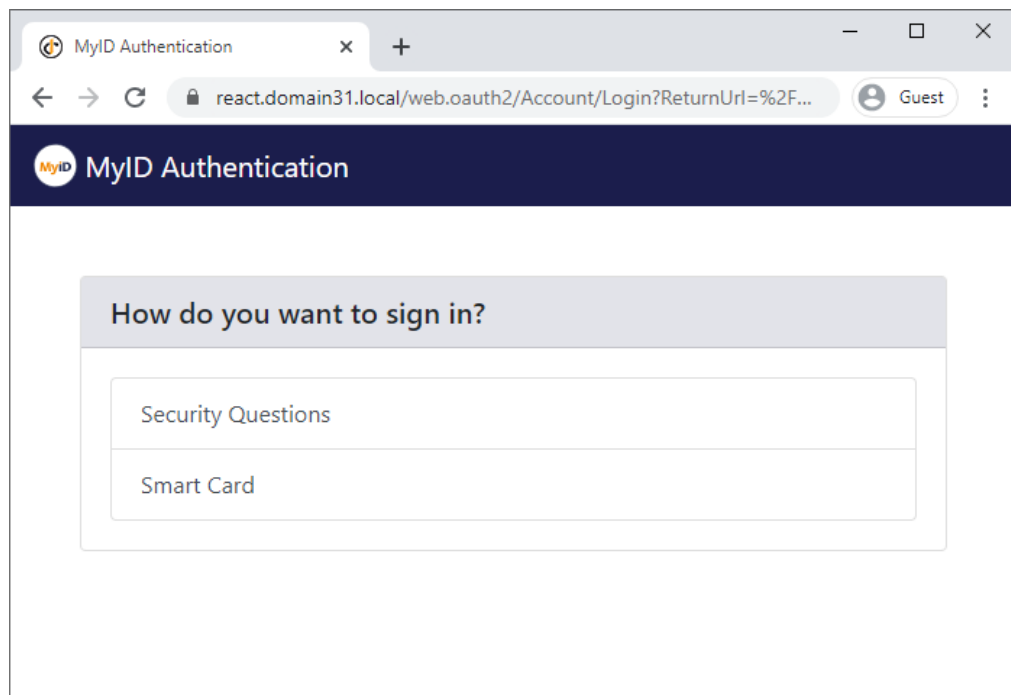
You must use the PKCE code challenge to request an authorization code from the MyID authentication service, and provide your user credentials.

1. From your website, post the following information to the MyID authorization URL:

`https://<server>/web.oauth2/connect/authorize`

- `client_id` – the ID of your system; for example:
`myid.myclient`
- `scope` – set this to `myid.rest.basic`
- `redirect_uri` – set this to the URL of the page on your website to which the authorization code will be returned. This must be the same as the URL you specified in the custom client configuration file.
- `response_type` – set this to `code`
- `code_challenge` – set this to the code challenge you generated. This is the Base64 URL-encoded SHA-256 hash of the random code verifier you created.
- `code_challenge_method` – set this to `S256`

When you post this request, the MyID authentication service prompts you for your user credentials. The available methods of authentication depend on how you have configured your system; the same methods are available for authentication as are available in the MyID Operator Client.



2. Complete the authentication using your method of choice; for example, security questions or smart card.

Note: You may need to ensure that the MyID Client Service is running on your PC.

3. Capture the `code` parameter that is returned to the page you specified in the `redirect_uri` parameter.

This is your authorization code, which can be used once to request an access token. You can use the access token repeatedly until it expires, but if you need to request another access token, you must first request another authorization code and go through the user authentication procedure again.

4.2.3 Requesting an access token

Once you have your authorization code, you can request an access token that allows you to call the API.

1. Post the following information to the MyID token URL:

`https://<server>/web.oauth2/connect/token`

- `grant_type` – set this to `authorization_code`
- `client_id` – the ID of your system; for example:
`myid.myclient`
- `code_verifier` – set this to the code verifier you created.
Note: Do not use the Base64 URL-encoded SHA-256 hash (the code challenge) – use the original plaintext value. The server compares this value to the encoded hash you provided when you requested the authorization code.
- `code` – set this to the authorization code you obtained from the server.
- `redirect_uri` – set this to the URL of the page on your website to which the access token will be returned. This must be the same as the URL you specified when you requested the authorization code.

2. Capture the `access_token` that is returned.

You can now use this access token to call the API.

See section [5.1, Calling the API from an external system](#) for more information on using an access token.

The request returns a block of JSON containing the following:

- `access_token` – your access token.
- `expires_in` – the lifetime in seconds for the token. Once the lifetime has expired, you must request a new access token.
- `token_type` – always `Bearer`.
- `scope` – the scope configured for the client in the `web.oauth2` web service; usually `myid.rest.basic`.

4.3 Obtaining an end-user based access token using a client secret

Once you have configured the web.oauth2 web service for a client secret, you can request your user-based authentication code, and use that to obtain an access token that you can use to call the API.

4.3.1 Requesting an authorization code

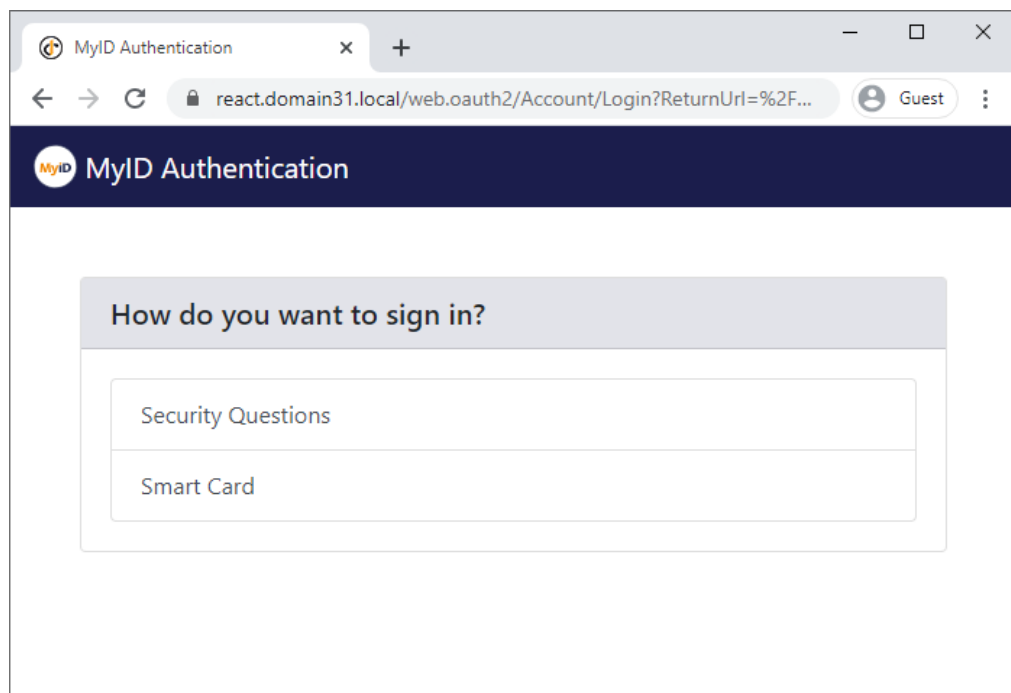
You must use the client secret to request an authorization code from the MyID authentication service, and provide your user credentials.

1. From your website, post the following information to the MyID authorization URL:

`https://<server>/web.oauth2/connect/authorize`

- `client_id` – the ID of your system; for example:
`myid.myclient`
- `scope` – set this to `myid.rest.basic`
- `redirect_uri` – set this to the URL of the page on your website to which the authorization code will be returned. This must be the same as the URL you specified in the custom client configuration file.
- `state` – this value is returned in the redirect, allowing you to persist data between the authorization request and the response; you can use this as a session key.
- `response_type` – set this to `code`

When you post this request, the MyID authentication service prompts you for your user credentials. The available methods of authentication depend on how you have configured your system; the same methods are available for authentication as are available in the MyID Operator Client.



2. Complete the authentication using your method of choice; for example, security questions or smart card.

Note: You may need to ensure that the MyID Client Service is running on your PC.

3. Capture the `code` parameter that is returned to the page you specified in the `redirect_uri` parameter.

This is your authorization code, which can be used once to request an access token. You can use the access token repeatedly until it expires, but if you need to request another access token, you must first request another authorization code and go through the user authentication procedure again.

4.3.2 Requesting an access token

Once you have your authorization code, you can request an access token that allows you to call the API.

1. Post the following information to the MyID token URL:

`https://<server>/web.oauth2/connect/token`

- `grant_type` – set this to `authorization_code`
- `client_id` – the ID of your system; for example:
`myid.myclient`
- `client_secret` – set this to the plaintext of the client secret you configured for the authentication service.

Note: Alternatively, you can combine the `client_id` and the `client_secret` and post them as a Basic authentication header; see section [3.3.1, Requesting an access token](#) for details.

- `code` – set this to the authorization code you obtained from the server.
- `redirect_uri` – set this to the URL of the page on your website to which the access token will be returned. This must be the same as the URL you specified when you requested the authorization code.

2. Capture the `access_token` that is returned.

You can now use this access token to call the API.

See section [5.1, Calling the API from an external system](#) for more information on using an access token.

The request returns a block of JSON containing the following:

- `access_token` – your access token.
- `expires_in` – the lifetime in seconds for the token. Once the lifetime has expired, you must request a new access token.
- `token_type` – always `Bearer`.
- `scope` – the scope configured for the client in the web.oauth2 web service; usually `myid.rest.basic`.

4.4 Using refresh tokens

You can configure the web.oauth2 authentication server to allow you to extend your authenticated session after obtaining the initial access token using a refresh token.

The process is as follows:

1. Configure the authentication server to allow refresh tokens.
See section [4.4.1, Configuring the authentication server](#).
2. Call the authentication `/connect/authorize` endpoint with a scope that allows refresh tokens.
See section [4.4.2, Obtaining the authorization code](#).
3. Call the authentication `/connect/token` endpoint and receive a refresh token in addition to the access token.
See section [4.4.3, Obtaining the access and refresh tokens](#).
4. If the access token has expired, or is close to expiry, call the authentication `/connect/token` endpoint with the refresh token to obtain a fresh access token and refresh token without having to re-authenticate.
See section [4.4.4, Using the refresh token to obtain a new access token](#).
5. Repeat the process of obtaining fresh access tokens and refresh tokens as often as required. You can use a refresh token up until its expiry (by default, after two hours) so if you are continually making calls to the API, and can request a fresh access token and refresh token every two hours, you do not need to authenticate until you hit the limit (by default, six days).

This is the same feature of the authentication server that is used for timeouts and re-authentication for the MyID Operator Client; for information about how this system works, see the *Timeouts and re-authentication* section in the [MyID Operator Client](#) guide.

4.4.1 Configuring the authentication server

You can configure the authentication server for end-user authentication using either PKCE or client secrets; see section [4.1.1, *Configuring the authentication service for PKCE*](#) or section [4.1.2, *Configuring the authentication service for a client secret*](#).

To allow the use of refresh tokens, you must make the following additional configuration changes:

1. In a text editor, open the file that contains the configuration for the client you are using.

Depending on how your system was configured, this may be:

- A custom client configuration file.

By default, these are located in the following folder:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

- The `appsettings.Production.json` file for the web service.

By default, this is:

```
C:\Program  
Files\Intercede\MyID\web.oauth2\appsettings.Production.json
```

This file is the override configuration file for the `appsettings.json` file for the web service. If this file does not already exist, you must create it in the same folder as the `appsettings.json` file.

2. In the client section for your API client, add the following settings:

- `AllowOfflineAccess` – set to `true` to allow refresh tokens.
- `SlidingRefreshTokenLifetime` – the number of seconds within which you can extend the authentication. The default is 7200 (two hours).
- `AbsoluteRefreshTokenLifetime` – the number of seconds after which you must re-authenticate, even if you have been continually extending the authentication. The default is 518400 (six days).

For example:

```
"Clients": [  
  {  
    "ClientId": "myid.mysystem",  
    "ClientName": "My External System",  
    "AllowOfflineAccess": true,  
    "SlidingRefreshTokenLifetime": 7200,  
    "AbsoluteRefreshTokenLifetime": 518400,  
    ...  
  }  
]
```

3. Save the configuration file.
4. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu

click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

4.4.2 Obtaining the authorization code

You can call the `/connect/authorize` endpoint to obtain an authorization code using either PKCE or a client secret; see section [4.2.2, *Requesting an authorization code*](#) (for PKCE) or section [4.3.1, *Requesting an authorization code*](#) (for client secrets).

When you request the authorization code, instead of requesting a scope of `myid.rest.basic`, request a scope of:

```
myid.rest.basic offline_access
```

This means that when you request an access token using the returned authorization code, it will additionally provide a refresh token.

4.4.3 Obtaining the access and refresh tokens

You can call the `/connect/token` endpoint to obtain an access token using either PKCE or a client secret; see section [4.2.3, *Requesting an access token*](#) (for PKCE) or section [4.3.2, *Requesting an access token*](#) (for client secrets).

If you have configured the authentication server to allow it, and added the `offline_access` scope to the request for the authorization code, the request returns a block of JSON containing the following:

- `access_token` – your access token.
- `expires_in` – the lifetime in seconds for the access token. Determined by the `AccessTokenLifetime` setting in the `web.oauth2` application settings file.
- `token_type` – always `Bearer`.
- `refresh_token` – your refresh token, which will be valid for the number of seconds determined by the `SlidingRefreshTokenLifetime` setting in the `web.oauth2` application settings file.
- `scope` – `myid.rest.basic offline_access`.

4.4.4 Using the refresh token to obtain a new access token

If the access token has expired, or is about to expire (which you can determine from the `expires_in` node of the returned JSON), you can use your refresh token to obtain a fresh access token.

1. Post the following information to the MyID token URL:

`https://<server>/web.oauth2/connect/token`

- `client_id` – the ID of your system; for example:
`myid.myclient`
- `grant_type` – set this to `refresh_token`
- `refresh_token` – set this to refresh token you obtained previously.

2. Capture the fresh `access_token` and `refresh_token` that are returned in the block of JSON.

You can now use this access token to call the API, and can use the new refresh token to obtain further access tokens as necessary.

Note: If the old access token has not yet expired, you can continue to use it; requesting a fresh access token does not invalidate the previous one. However, you can use a refresh token only once.

If your access token has expired and your refresh token has expired, or if you have exceeded the limit since the original authorization code was requested (as determined by the `AbsoluteRefreshTokenLifetime` setting in the `web.oauth2` application settings file) you must call `/connect/authorize` again to re-authenticate.

If the refresh token has expired, or if the `AbsoluteRefreshTokenLifetime` limit has been exceeded, a response of `invalid_grant` is returned.

4.5 Revoking access tokens

You can revoke an access token for the web.oauth2 authentication server; the server supports the OAuth2/OpenID Connect revocation endpoint.

Where the token contains a scope that allows accessing the MyID Core API, the revocation endpoint updates the MyID database to ensure that the access token can no longer be used.

If you are using refresh tokens, these are also invalidated.

Note: Revoking a token that does not include a scope that relates to the MyID database (for example, if it does not have `myid.rest.basic` scope because the token is used for products outside of MyID) invalidates any refresh tokens, but the access token remains valid until expiry. This is because an access token is valid until expiry, *unless* there is a back-end system that can be updated to indicate that access token is no longer valid.

To call the revocation endpoint, post the following information to the MyID revocation URL, formatted according to the RFC for OAuth 2.0 Token Revocation (RFC 7009):

`https://<server>/web.oauth2/connect/revocation`

- `client_id` – the ID of your system; for example:
`myid.myclient`
- `token` – set this to the access token.
- `token_type_hint` – set this to:
`access_token`

5 Calling the API

You must configure your server for the appropriate method of authentication.

- To configure the server and obtain an access token for server-to-server authentication, see section 3, [Server-to-server authentication](#).
- To configure the server and obtain an access token for user authentication, see section 4, [End-user authentication](#).

Once you have obtained an access token, you can call the API from an external system; see section 5.1, [Calling the API from an external system](#).

You can also call the API from within the Swagger-based documentation; see section 5.2, [Calling the API from the documentation](#).

5.1 Calling the API from an external system

Once you have obtained an access token, you can call the API.

Present the access token in an `Authorization` header with a type of `Bearer` (as detailed in the `token_type` option in the returned JSON containing your access token).

Important: You must keep track of the lifetime of the token, and request a new token when the current token has expired. Avoid obtaining a new access token if the previous token is still valid, as this generates unnecessary records in the `Logons` table in the MyID database, and may impact your system performance.

For example, to view the details of a particular person, you use the following:

```
GET /api/People/{id}
```

This method requires a parameter `dirInfo` and the ID of the person – this example uses the following ID:

```
53F2A29B-376B-4600-867C-2E5BD95AE222
```

See the API documentation for the particular requirements for each method.

For example, using cURL:

```
curl -k -i -X GET "https://myserver.example.com/rest.core/api/People/53F2A29B-376B-4600-867C-2E5BD95AE222" -d "dirInfo=true" -H "Authorization: Bearer <token string>"
```

or using PowerShell:

```
# Access token for the API
$token = "<token string>"

# Set the body of the request. The parameters depend on the method being used;
# see the Swagger-based API documentation for details.
$body = @{dirInfo='true'
}

# Set the headers for the request
$header = @{Authorization="Bearer $token"
}

# Call the method. This example obtains the information about a particular
```

```
# person. The user configured for the API must have access to this feature
# through their role assignment, and must have scope that allows them to
# view the details of the specified person.
Invoke-WebRequest -Method GET -Uri
'https://myserver.example.com/rest.core/api/People/53F2A29B-376B-4600-867C-2E5BD95AE222' -
body $body -Headers $header | Select-Object -Expand Content

# Wait for a keypress
Write-Host "`r`nPress any key to continue..." -ForegroundColor Yellow

[void][System.Console]::ReadKey($true)
```

5.2 Calling the API from the documentation

The Swagger-based documentation for the API allows you to try the methods from within the documentation itself, once you have configured the server for the appropriate method of authentication.

5.2.1 Calling the API from the documentation using server-to-server authentication

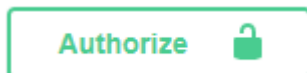
Once you have configured the web server for server-to-server authentication, you can also use the client ID and shared secret to authenticate to the server to access the API features from the Swagger-based API documentation:

1. Open the API documentation in a browser.

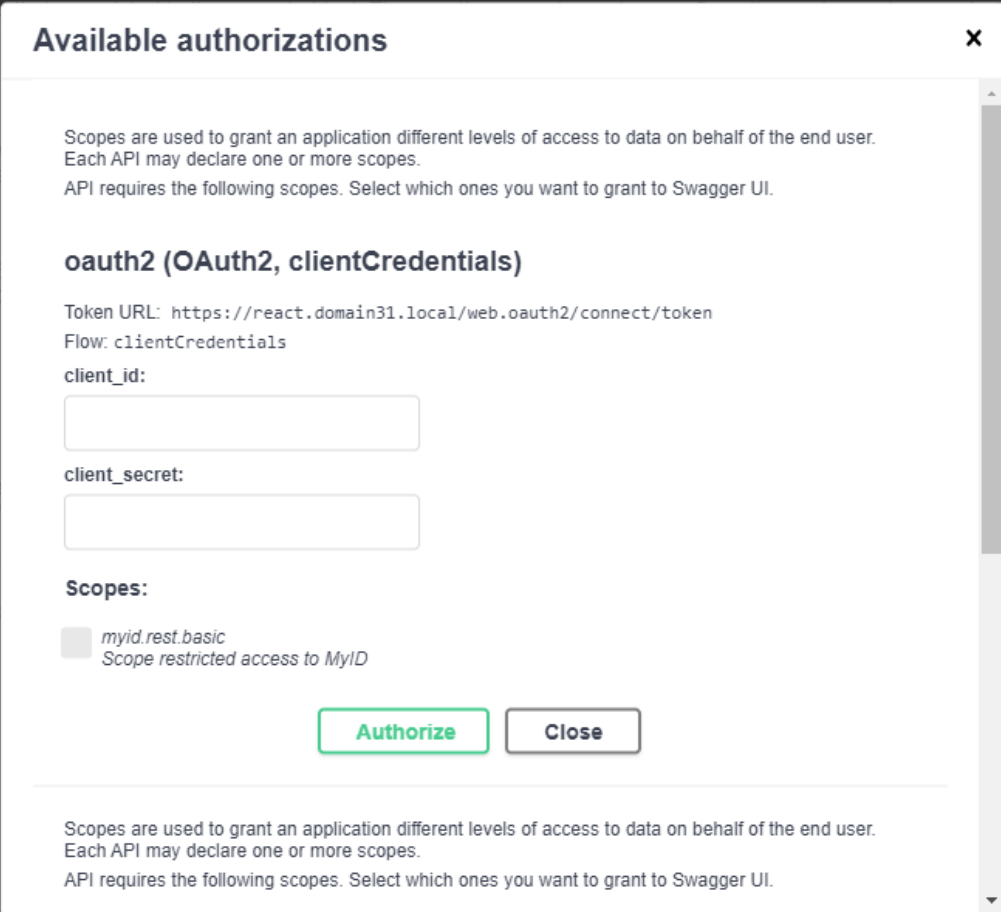
See section [2.1, Accessing the API documentation](#) for details.

You must set the `AllowTestFromDocumentation` option to `true` to allow you to use the documentation as a test harness.

2. Click **Authorize**.



The Available authorizations screen appears.



Available authorizations ✕

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes. API requires the following scopes. Select which ones you want to grant to Swagger UI.

oauth2 (OAuth2, clientCredentials)

Token URL: `https://react.domain31.local/web.oauth2/connect/token`
Flow: `clientCredentials`

client_id:

client_secret:

Scopes:

☒ `myid.rest.basic`
Scope restricted access to MyID

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes. API requires the following scopes. Select which ones you want to grant to Swagger UI.

3. In the **oauth2 (OAuth2, clientCredentials)** section, enter the following:
 - **client_id** – your client ID; for example, `myid.mysystem`.
 - **client_secret** – your client secret; for example:
`82564d6e-c4a6-4f64-a6d4-cac43781c67c`
Note: Do not use the combined client ID and Base64 version of the client secret.
 - **Scopes** – select the **myid.rest.basic** option.
4. Click **Authorize**.
5. Click **Close**.

5.2.2 Calling the API from the documentation using end-user authentication

You can use end-user authentication to access the API from within the documentation.

1. Edit the `appsettings.json` file for the `web.oauth2` web service to include the Swagger callback URL.
 - a. Open the `appsettings.json` file in a text editor.

By default, this is:

`C:\Program Files\Intercede\MyID\web.oauth2\appsettings.json`

Note: If you have an `appsettings.Production.json` file, use that instead. This is the override configuration file for the `appsettings.json` file for the web service.

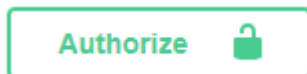
- b. In the **Clients** section, under the `myid.operatorclient` settings, add the following to the `RedirectUri` array:

```
https://<server>/rest.core/swagger/oauth2-redirect.html
```

where `<server>` is the name of your MyID authentication. For example:

```
"RedirectUri": [  
  "https://react.domain31.local/MyID/OperatorClient",  
  "https://react.domain31.local/rest.core/swagger/oauth2-  
  redirect.html"  
]
```

2. Click **Authorize**.



3. The Available authorizations screen appears.

Available authorizations

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes. API requires the following scopes. Select which ones you want to grant to Swagger UI.

oauth2 (OAuth2, clientCredentials)

Token URL: `https://react.domain31.local/web.oauth2/connect/token`
Flow: `clientCredentials`

client_id:

client_secret:

Scopes:

☐ `myid.rest.basic`
Scope restricted access to MyID

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes. API requires the following scopes. Select which ones you want to grant to Swagger UI.

4. Scroll to the **oauth2 (OAuth2, authorizationCode)** section, and enter the following:

- **client_id** – enter `myid.operatorclient`.

If you have set up your own system to use end-user authentication, you can use this client ID instead.

- **Scopes** – select the **myid.rest.basic** option.

5. Click **Authorize**.
6. Complete the authentication using your method of choice; for example, security questions or smart card.

Note: You may need to ensure that the MyID Client Service is running on your PC.

7. Click **Close**.

6 Operation extension

You can extend the authorization for your API calls to call additional operations through the MyID Client Service; this allows you to carry out some operations (for example, resetting a device PIN) by launching MyID Desktop; you can use this feature to carry out actions that are not yet supported by the MyID Core API. You can also carry out operations provided by the MyID Client Service; for example, using the Select Security Device dialog to select a device.

You can carry out the following:

- Using the MyID Client Service API to launch an operation.
See section [6.1, Obtaining an operation extension token](#).
- Using the MyID Client Service API to open the Select Security Device dialog.
section [6.2, Obtaining an extension token for Select Security Device](#).

The MyID Client Service API documentation is available in the MyID Integration Toolkit, available on request.

6.1 Obtaining an operation extension token

Before you can call the MyID Client Service API to launch MyID Desktop, you must obtain an operation extension token for this particular operation on this particular object (for example, device, request, or person) – this is a short-lived authorization code for a single use.

When you call the MyID Core API, the block of data returned contains a series of links. If a link has a `cat` of `myid.mcs`, this means you can carry out this operation using the MyID Client Service.

For example:

```
{
  "op": "297",
  "cat": "myid.mcs",
  "desc": "Reset Card PIN",
  "verb": "",
  "auth": "client_id=myid.mcs&grant_type=operation&op=297&scope=myid.operation&deviceId=<DEVICE ID>&token={token}",
  "clientData": ["DSK"]
}
```

You can use this information to obtain an operation extension token, then use this token to call the MyID Client Service API to launch the workflow.

To request the operation extension:

1. Post the following information to the MyID token URL:

`https://<server>/web.oauth2/connect/token`

- `client_id` – set this to `myid.mcs`
- `grant_type` – set this to `operation`
- `op` – set this to the ID of the operation you want to carry out.

For example:

- Operation ID 297 is **Reset PIN**.
- Operation ID 5007 is **Assisted Activation**.

You can obtain the operation ID from the link data.

- `scope` – set this to `myid.operation`
- `deviceId`, `personId`, or `requestId` – provide the ID of a single device, person, or request. You can obtain the ID from the link data.
- `token` – set this to your existing authorization token.

See section 3.3, *Obtaining a server-to-server access token*, section 4.2, *Obtaining an end-user based access token using PKCE*, or section 4.3, *Obtaining an end-user based access token using a client secret* for details.

2. Capture the access token that is returned.

You can now use this access token in the `Token` argument of the `StartWithToken` method of the MyID Client Service API to launch MyID Desktop for the specified operation and target.

6.2 Obtaining an extension token for Select Security Device

Before you can call the MyID Client Service API to open the Select Security Device dialog authenticated with the logged-on operator, you must obtain an extension token for this particular operation – this is a short-lived authorization code for a single use.

Note: This authenticated mode provides user images and full names on the smart card selection screen based on the scope and administration groups of the logged-on user. If you do not need to display this additional detail, you can call the `SelectCard` method of the MyID Client Service API without the `Token` parameter; in this case, you do not need to obtain an extension token.

To use authenticated mode, you must ensure that the MyID web.oauth2 server is configured to allow a scope of `myid.devicepicker`. Check the `appsettings.json` file (by default, in the `C:\Program Files\Intercede\MyID\web.oauth2\` folder) for the following:

- In the `Scopes` array, a scope called `myid.devicepicker` exists, with the following `UserClaims`:
 - `deviceDetectContext`
 - `op`
 - `myidSessionId`

- In the `ApiResources` array, for the resource with name `myid.mws`, the scope `myid.devicepicker` is in the `Scopes` array.
- In the `Clients` array, for the client with ID `myid.mcs`, the scope `myid.devicepicker` is in the `AllowedScopes` array.

To obtain the extension token:

1. Post the following information to the MyID token URL:

`https://<server>/web.oauth2/connect/token`

- `client_id` – set this to `myid.mcs`
- `grant_type` – set this to `operation`
- `op` – set this to the ID of the **Read Card (Authenticated)** operation. This is `100221`.
- `scope` – set this to `myid.devicepicker`
- `token` – set this to your existing authorization token.

See section [3.3, Obtaining a server-to-server access token](#), section [4.2, Obtaining an end-user based access token using PKCE](#), or section [4.3, Obtaining an end-user based access token using a client secret](#) for details.

- `deviceDetectContext` – reserved for future use. Leave as an empty query parameter.

2. Capture the access token that is returned.

You can now use this access token in the `Token` argument of the `SelectCard` method of the MyID Client Service API to launch the Select Security Device dialog authenticated with the logged-on operator.

7 Managing directories

You can use the MyID Core API to manage your directories.

For example, you may want to update the passwords for all of your directories. You can write a script that iterates through your directories, tests the connection for each directory using the new password, then, if that succeeds, updates the directory configuration with the new password, then finally verifies that MyID can still connect to the directory.

The API provides features that allow you to:

- Get a list of directories, including full details of each directory (excluding the password):

```
GET /api/Dirs
```

- Get full details for a specific directory (excluding the password):

```
GET /api/Dirs/{directoryId}
```

- Test the connection to the directory using new credentials:

```
POST /api/Dirs/{directoryId}/test
```

- Update the settings for a directory:

```
PATCH /api/Dirs/{directoryId}/update
```

- Verify the connection to the directory using the existing credentials:

```
POST /api/Dirs/{directoryId}/verify
```

By default, users with access to **Directory Management** in the **Edit Roles** workflow have access to these endpoints.

For full information on using the API endpoints, including the available parameters and permissions, see the `Dirs` section in the API documentation; see section [2.1, Accessing the API documentation](#) for details of viewing the Swagger API documentation.

MyID also provides a sample PowerShell script that demonstrates managing your directories through the API; see section [7.1, Sample PowerShell script for managing directories](#).

For information on possible errors that may occur when managing directories through the MyID Core API, see section [7.2, Troubleshooting managing directories](#).

7.1 Sample PowerShell script for managing directories

Your MyID release package includes a sample PowerShell script that demonstrates the use of the MyID Core API to manage directories.

This script is provided in the following location in the MyID release package:

```
\Support Tools\MyID Core API Sample Scripts\Directory Management\
```

7.1.1 Setting up the web.oauth2 client for the script

The sample script uses a new client called `directory.connection.demo` – you must add this client to the configuration of the web.oauth2 service.

First, create a custom client configuration file.

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomClients` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the client ID, but by the index of the entry in the `Clients` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom clients, but to use separate files in the `CustomClients` folder instead.

For more information about custom configuration files, see section 9, [Custom configuration files](#).

Now that you have a custom client configuration file, you can edit it to add your client details:

```
{
  "ClientId": "directory.connection.demo",
  "ClientName": "Directory Connection Demo",
  "AccessTokenLifetime": 3600,
  "AllowedGrantTypes": [
    "client_credentials"
  ],
  "ClientSecrets": [
    {
      "Value": "qQppgDNaU5AgBcoADy/MtWx7ieJ3CAigystWLBn32es="
    }
  ],
  "AllowedScopes": [
    "myid.rest.basic"
  ]
}
```



```
  ],  
  "Properties": {  
    "MyIDLogonName": "dirconn.demo"  
  }  
}
```

Important: The sample script uses a fixed, unencrypted client secret. You must not use this script on a production system. See section [3.2, *Configuring web.oauth2 for server-to-server authentication*](#) for details of creating your own client secret. On a production system, you are recommended to use DPAPI to encrypt the client secret stored in your scripts; for an example of doing this, see section [10.3.4, *Encrypting the client secret*](#).

7.1.2 Refreshing the authentication server settings

Once you have made your changes to the web.oauth2 server settings, you must refresh the application pool to ensure that all the systems are using the latest settings.

To refresh the server settings:

1. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

2. Check that the web.oauth2 server is still operational by logging on to the MyID Operator Client.

Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

```
HTTP Error 500.30 - ANCM In-Process Start Failure
```

7.1.3 Setting up the user account for the script

The sample script uses a MyID user called `dirconn.demo` – you must add a user with this logon name to your MyID system, and grant the user a role with access to the **Directory Management** option in the **Edit Roles** workflow.

You can use the `ShowPermissions` option to display the roles that have access to the `Dirs` endpoints in the API documentation; see section [2.1, *Accessing the API documentation*](#) for details.

7.1.4 Editing the script to use your own server address

Copy the `DirectoryConnectionDemo.ps1` file from the following location in the MyID release package:

```
\Support Tools\MyID Core API Sample Scripts\Directory Management\
```

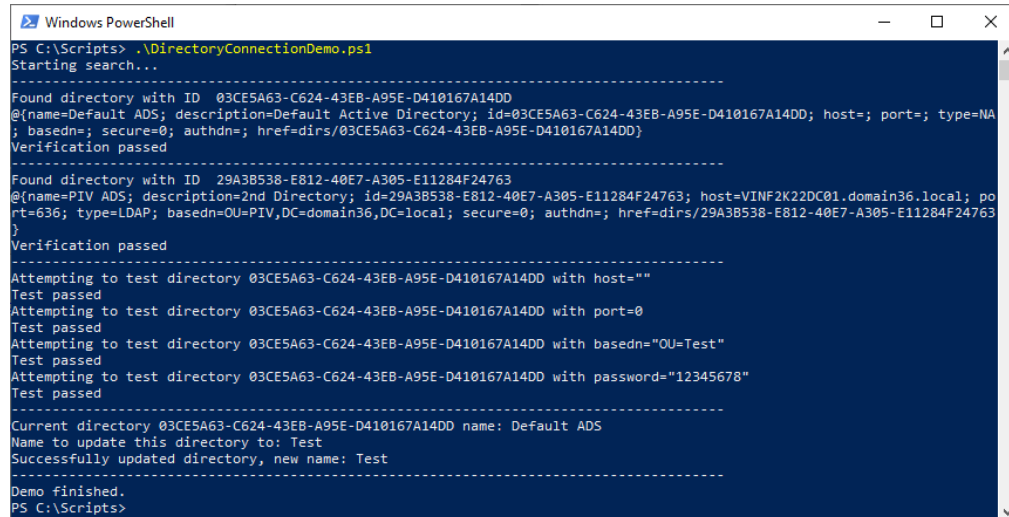
In a text editor, open the script.

The script uses a server address of `react.domain36.local` throughout. You must change each instance to the web address of your own MyID server.

7.1.5 Running the script

Open a Windows PowerShell command prompt and run the modified script from where you saved it to:

```
.\DirectoryConnectionDemo.ps1
```



```
Windows PowerShell
PS C:\Scripts> .\DirectoryConnectionDemo.ps1
Starting search...
-----
Found directory with ID 03CE5A63-C624-43EB-A95E-D410167A14DD
@{name=Default ADS; description=Default Active Directory; id=03CE5A63-C624-43EB-A95E-D410167A14DD; host=; port=; type=NA
; basedn=; secure=0; authdn=; href=dirs/03CE5A63-C624-43EB-A95E-D410167A14DD}
Verification passed
-----
Found directory with ID 29A3B538-E812-40E7-A305-E11284F24763
@{name=PIV ADS; description=2nd Directory; id=29A3B538-E812-40E7-A305-E11284F24763; host=VIN2K22DC01.domain36.local; po
rt=636; type=LDAP; basedn=OU=PIV,DC=domain36,DC=local; secure=0; authdn=; href=dirs/29A3B538-E812-40E7-A305-E11284F24763
}
Verification passed
-----
Attempting to test directory 03CE5A63-C624-43EB-A95E-D410167A14DD with host=""
Test passed
Attempting to test directory 03CE5A63-C624-43EB-A95E-D410167A14DD with port=0
Test passed
Attempting to test directory 03CE5A63-C624-43EB-A95E-D410167A14DD with basedn="OU=Test"
Test passed
Attempting to test directory 03CE5A63-C624-43EB-A95E-D410167A14DD with password="12345678"
Test passed
-----
Current directory 03CE5A63-C624-43EB-A95E-D410167A14DD name: Default ADS
Name to update this directory to: Test
Successfully updated directory, new name: Test
-----
Demo finished.
PS C:\Scripts>
```

The script runs through the following stages:

1. Gets a list of each directory in your system, displays its details, and verifies the connection.
2. Runs a series of tests against the first directory with a variety of changes.
3. Offers a prompt to allow you to change the name of the first directory.

Note: When you change the name of the directory, to confirm the change, you can recycle the **myid.rest.core.pool** application pool in IIS on the MyID web server, log out of the MyID Operator Client, then log back in again and see the change in the **People** category.

7.2 Troubleshooting managing directories

The following errors may occur when managing directories through the MyID Core API.

- WS40068 – The directory connection failed with the supplied credentials.
- WS40069 – Supplied credentials cannot build a valid server location.

See the *MyID Operator Client error codes* section in the [Error Code Reference](#) guide for potential causes and solutions.

8 Reassigning devices

You can use the MyID Core API to reassign devices from one person to another. This feature is available only through the API, not through the MyID Operator Client.

The API provides the following endpoint:

- `POST /api/Devices/{id}/reassign`

Users with access to the **Reassign Device** option in the **Cards** section of the **Edit Roles** workflow have access to this endpoint.

For full information on using this API endpoint, including details of the available parameters and permissions, see the [Devices](#) section in the API documentation; see section [2.1, Accessing the API documentation](#) for details of viewing the Swagger API documentation.

This endpoint allows you to specify a device by its ID. In the body of the call, you provide the ID of the target person to whom you want to assign the device:

```
{
  "target": {
    "id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
  }
}
```

Note: The user account calling the API must have scope over both the current owner and the new owner.

The device must:

- be fully issued and active.
- be issued with a credential profile that has **Contact** as the **Card Encoding**.
- not contain additional identity certificates.
- not be issued with a credential profile that has the **Key Recovery Only** option set.
- not already be issued to the target person.

If the request is successful, MyID:

- Returns a block of JSON containing details about the device, including its new owner.
- Transfers the ownership of the device, and its certificates, in the MyID database to the new owner.

Note: The physical device is unaffected by this change. Only the MyID database is updated.

- Cancels any existing requests for the device.
- Triggers a REST notification.

See the *REST Device Reassigned notification* in the [REST Web Service Notifications](#) guide for details of this notification.

If the request fails, it returns an error message providing details of the failure.

cURL, Python, and PowerShell examples are provided; see section [8.1, Examples for reassigning a device](#).

For errors that may occur when attempting to reassign a device, see section [8.2](#), *Troubleshooting reassigning devices*.

8.1 Examples for reassigning a device

Assume you have a device with the following ID:

B0BF0B14-CA1E-415A-AC4C-AB8F23BC390B

It is currently assigned to User A, but you want to reassign it to User B, who has the ID:

B2573666-2748-489C-A492-D19A88C7CE10

These examples assume your server is on `myserver.example.com`, and that you have already obtained an access token; `<YOUR-TOKEN>` is used as a placeholder.

8.1.1 cURL

```
curl -X "POST" "https://myserver.example.com/rest.core/api/Devices/B0BF0B14-CA1E-415A-AC4C-AB8F23BC390B/reassign" -H "accept: application/json" -H "x-api-version: 1" -H "Content-Type: application/json-patch+json" -H "Authorization: Bearer <YOUR TOKEN>" -d '{"target": { "id": "B2573666-2748-489C-A492-D19A88C7CE10" }}
```

8.1.2 Python

```
import requests
import json

# Set the server
server = "myserver.example.com"

# Set the ID of the device you want to reassign
deviceID = "B0BF0B14-CA1E-415A-AC4C-AB8F23BC390B"

# Set the ID of the person to whom you want to reassign the device
targetID = "B2573666-2748-489C-A492-D19A88C7CE10"

# Get the access token
token = "<YOUR-TOKEN>"

# Create the payload for the API call containing the ID of the target person
targetData = {
    "target": {
        "id": targetID
    }
}
target = json.dumps(targetData)

# Set up the call for the API
response = requests.post(
    "https://" + server + "/rest.core/api/Devices/" + deviceID + "/reassign",
    headers={"Authorization": "Bearer " + token,
            "Content-Type": "application/json-patch+json",
            "accept": "application/json",
            "x-api-version": "1"},
    data=target)

# Display the response
if response.status_code==200:
    print("The device has been reassigned to:")
    returnedData = json.loads(response.text)
    print(returnedData["owner"]["name"])
else:
    print("An error occurred:")
    returnedData = json.loads(response.text)
    print("Error code: " + returnedData["code"])
    print("Error message: " + returnedData["message"])
```

8.1.3 PowerShell

```
# Set the server
$server = "myserver.example.com"

# Set the ID of the device you want to reassign
$deviceID = 'B0BF0B14-CA1E-415A-AC4C-AB8F23BC390B'

# Set the ID of the person to whom you want to reassign the device
$targetID = 'B2573666-2748-489C-A492-D19A88C7CE10'

# Get the access token
$token = "<YOUR-TOKEN>"

# Create the payload for the API call containing the ID of the target person
$targetData = '{"target': {'id': '$targetID'}}"

# Set up the call for the API
$authHeader = @{
    'Content-Type'='application/json-patch+json'
    'Authorization'='Bearer $token'
    'x-api-version'=' 1'
}
$URI = 'https://' + $server + '/rest.core/api/Devices/' + $deviceID + '/reassign'
$reassignRequest = @{
    Headers = $authHeader
    Uri = $URI
    Method = "POST"
    Body = $targetData
}

# Display the response
try {
    $result = Invoke-WebRequest @reassignRequest | ConvertFrom-Json
    Write-Host "The device has been reassigned to:"
    Write-Host $result.owner.name
}
catch {
    $result = $_.Exception.Response.GetResponseStream()
    $reader = New-Object System.IO.StreamReader($result)
    $reader.BaseStream.Position = 0
    $reader.DiscardBufferedData()
    $responseBody = $reader.ReadToEnd() | ConvertFrom-Json
    Write-Host "An error occurred:"
    Write-Host "Error code:" $responseBody.code
    Write-Host "Error message:" $responseBody.message
}
```

8.2 Troubleshooting reassigning devices

The following errors can occur when attempting to reassign a device:

- WS40074 – Must select a new owner to reassign.
- WS50088 – The device has to be actively issued to change owner.
- WS50089 – Cannot reassign device with additional identity certificates on it.
- WS50090 – Cannot reassign device with a credential profile marked as key recovery.
- WS50091 – Cannot reassign device when the credential profile does not have an encoding capability of type contact.

See the *MyID Operator Client error codes* section in the [Error Code Reference](#) guide for potential causes and solutions.

9 Custom configuration files

When configuring your web.oauth2 web service to add custom clients, scopes, API resources, and identity resources to allow you to work with the MyID Core API, you can edit arrays in the `appsettings.Production.json` file for the web.oauth2 web service. Because elements in this array are determined by their index, you must pad the array to the correct size to correspond to the entries in the `appsettings.json` file and prevent accidentally overwriting other clients.

To simplify this process, and to allow for greater maintainability, you are recommended instead to create separate configuration files for each client, scope, API resource, or identity resource.

You can create custom files for the following:

- Clients.
See section [9.1, Custom client configuration files](#).
- Scopes.
See section [9.2, Custom scope configuration files](#).
- API resources.
See section [9.3, Custom API resource configuration files](#).
- Identity resources.
See section [9.4, Custom identity resource configuration files](#).

9.1 Custom client configuration files

Custom client configuration files each contain the details for a single client, which is identified by its `ClientID`. If the `ClientID` already exists in the `appsettings.Production.json` or `appsettings.json` file, the details from the custom client configuration file completely replace the existing settings for that client; if the `ClientID` does not exist, the client is added to the configuration for the web service.

Note: You can continue to use the `appsettings.Production.json` file for your client configuration if you want, but you are recommended to move any clients you have created into their own custom client configuration files.

To create a custom client configuration file:

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomClients` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the client ID, but by the index of the entry in the `Clients` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom clients, but to use separate files in the `CustomClients` folder instead.

9.2 Custom scope configuration files

Custom scope configuration files each contain the details for a single scope, which is identified by its `Name`. If the `Name` already exists in the `appsettings.Production.json` or `appsettings.json` file, the details from the custom scope configuration file completely replace the existing settings for that scope; if the `Name` does not exist, the scope is added to the configuration for the web service.

Note: You can continue to use the `appsettings.Production.json` file for your scope configuration if you want, but you are recommended to move any scopes you have created into their own custom scope configuration files.

To create a custom scope configuration file:

1. On the MyID web server, navigate to the `CustomScopes` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomScopes\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your scope configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the scope as the filename.

You can create a custom `.json` file for each scope that you want to add. You can include only one scope in each file, but you can have multiple files if you need multiple scopes. These scopes are added to the `Scopes` array from the `appsettings.json` file. You must use a unique `Name`; if you use the same `Name` in a custom file as an already existing scope in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomScopes` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the `Name`, but by the index of the entry in the `Scopes` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom scopes, but to use separate files in the `CustomScopes` folder instead.

9.3 Custom API resource configuration files

Custom API resource configuration files each contain the details for a single API resource, which is identified by its `Name`. If the `Name` already exists in the `appsettings.Production.json` or `appsettings.json` file, the details from the custom API resource configuration file completely replace the existing settings for that API resource; if the `Name` does not exist, the API resource is added to the configuration for the web service.

Note: You can continue to use the `appsettings.Production.json` file for your API resource configuration if you want, but you are recommended to move any API resources you have created into their own custom API resource configuration files.

To create a custom API resource configuration file:

1. On the MyID web server, navigate to the `CustomAPIResources` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomAPIResources\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your API resource configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the API resource as the filename.

You can create a custom `.json` file for each API resource that you want to add. You can include only one API resource in each file, but you can have multiple files if you need multiple API resources. These API resources are added to the `APIResources` array from the `appsettings.json` file. You must use a unique `Name`; if you use the same `Name` in a custom file as an already existing API resource in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomAPIResources` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the `Name`, but by the index of the entry in the `APIResources` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom API resources, but to use separate files in the `CustomAPIResources` folder instead.

9.4 Custom identity resource configuration files

Custom identity resource configuration files each contain the details for a single identity resource, which is identified by its `Name`. If the `Name` already exists in the `appsettings.Production.json` or `appsettings.json` file, the details from the custom identity resource configuration file completely replace the existing settings for that identity resource; if the `Name` does not exist, the identity resource is added to the configuration for the web service.

Note: You can continue to use the `appsettings.Production.json` file for your identity resource configuration if you want, but you are recommended to move any identity resources you have created into their own custom identity resource configuration files.

To create a custom identity resource configuration file:

1. On the MyID web server, navigate to the `CustomIdentityResources` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomIdentityResources\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your identity resource configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the identity resource as the filename.

You can create a custom `.json` file for each identity resource that you want to add. You can include only one identity resource in each file, but you can have multiple files if you need multiple identity resources. These identity resources are added to the `IdentityResources` array from the `appsettings.json` file. You must use a unique `Name`; if you use the same `Name` in a custom file as an already existing identity resource in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomIdentityResources` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the `Name`, but by the index of the entry in the `IdentityResources` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom identity resources, but to use separate files in the `CustomIdentityResources` folder instead.

10 Example – server-to-server

This chapter contains a worked example of calling the MyID Core API using the server-to-server authentication method.

This worked example provides details of configuring your MyID server, then using Python scripts to obtain an access token and call the API. It then goes on to provide example scripts that take you from getting simple results from the MyID database to adding a person, requesting a device, and checking the status of the request. Further examples are provided for getting multiple-page reports and working with access tokens.

By the time you complete these examples, you should have a basic understanding of how to authenticate to the server and call the API to obtain and post information to and from the server.

Important: This chapter contains code samples that you can copy and paste into your own project. Due to the limitations of PDFs and HTML pages, some whitespace characters may not come across cleanly; this can cause issues with JSON files particularly. You must also ensure that the indentation on the Python code samples is maintained; Python uses indentation to delineate code blocks. You are recommended to validate and sanitize your code samples before using them; for example, you can use a JSON reformatter to ensure that the JSON files have the correct format.

Some example scripts rely on you having completed earlier stages. If you want to go through these worked examples, it is recommended that you work through them in order.

10.1 Requirements

This example has been developed using Python on Windows; however, the code samples have been written to aid readability and portability to other languages and environments.

If you want to follow and run the code samples in this chapter, you require the following:

- Python for Windows.

These examples were developed using Python 3.12.1.

Python for Windows is available for download from the Python website:

www.python.org/downloads/windows/

The examples were developed using Visual Studio Code with the Python extension. If you are unfamiliar with Python, you may want to use this code editor, as it provides a helpful code completion feature and debugging support.

code.visualstudio.com

- The following additional Python modules:

- `requests`
- `pip-system-certs`
- `pywin32`
- `pyjwt`
- `cryptography`

You can install the modules as required (for example, if you do not want to carry out the JWT decoding example, you do not need `pyjwt` or `cryptography`), or you can install all the modules before beginning:

```
py -m pip install requests
py -m pip install pip-system-certs
py -m pip install pywin32
py -m pip install pyjwt
py -m pip install cryptography
```

Note: You may have to restart your code editor after installing new modules.

- An installed and operational MyID system.

You are recommended to run these examples in a pre-production environment.

Configuring the required permissions is easier if you have the startup user account and Startup User role available. On a production system, you can restrict the permissions of the API credentials to the minimum necessary to carry out your tasks.

You must have:

- Access to the MyID web server with permission to edit configuration files and recycle application pools in IIS.
- The ability to create new MyID user accounts that have permission to access any workflow.

10.2 Creating a user account

You must select a role, group, and user account for your scripts to use to call the MyID Core API. The role permissions of the user account determine which features of the API you can call, and the scope and group determine which user accounts you can work with.

See section [3.1, *Configuring MyID for server-to-server authentication*](#) for details of configuring MyID for a user account to call the API.

For a pre-production system, you can use the System Startup group and the Startup User role; however, you are still recommended to create a separate user account. These examples use an account with the logon name:

```
api.external
```

For evaluation purposes, make sure your new user account has permission to access *every* feature in MyID. Use the **Edit Roles** workflow in MyID Desktop to grant permission to every feature to the role you are using for the API user; by default, the Startup User role has access to most, but not all, features. For production systems, you can lock down the user account to be able to access only the necessary features; see section [2.2, *Accessing the API features*](#).

10.3 Authenticating to the MyID Core API

The MyID Core API (rest.core) requires authentication through the MyID web.oauth2 service. You must configure the web.oauth2 service to add a new client to be used by your scripts, with a client secret that is used to secure the connection. Once you have done that, you can encrypt the client secret, and configure your scripts to authenticate to the MyID Core API.

10.3.1 Generating a client secret

You must produce a client secret and generate a Base64-encoded SHA-256 hash of this secret.

You can use the provided `GenClientSecret.ps1` PowerShell script to create a new secret and generate a Base64-encoded SHA-256 hash.

To generate a client secret and hash:

1. On the MyID web server, open a Windows PowerShell command prompt.
2. Navigate to the web.oauth2 folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\
```

3. Run the following:

```
.\GenClientSecret.ps1
```

The script displays its output on screen. For example:

```
client secret: b5989015-bb9e-4533-874b-2b4a6a8280ed
```

```
SHA256+base64: Pxvo4ww2jQ1wW0RsnL9q79TcC/f0b57P07os0ZU7Frs=
```

4. Take a note of the output.

You need the `SHA256+base64` value for the client section in the custom client configuration file for the web.oauth2 web service; see section [10.3.2, Adding the API client](#).

You must encrypt the `client secret` before you can use it in your scripts; see section [10.3.4, Encrypting the client secret](#).

10.3.2 Adding the API client

The sample API scripts use a new client called `myid.mysystem` – you must add this client to the configuration of the `web.oauth2` service.

First, create a custom client configuration file.

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomClients` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the client ID, but by the index of the entry in the `Clients` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom clients, but to use separate files in the `CustomClients` folder instead.

For more information about custom configuration files, see section 9, [Custom configuration files](#).

Now that you have a custom client configuration file, you can edit it to add your client details:

```
{
  "ClientId": "myid.mysystem",
  "ClientName": "My External System",
  "AccessTokenLifetime": "3600",
  "AllowedGrantTypes": [
    "client_credentials"
  ],
  "ClientSecrets": [
    {
      "Value": "<hash of secret>"
    }
  ],
  "AllowedScopes": [
    "myid.rest.basic"
  ]
}
```

```
    ],  
    "Properties": {  
      "MyIDLogonName": "<user account>"  
    }  
  }  
}
```

Where:

- `<hash of secret>` is the Base64-encoded SHA-256 hash of the client secret you generated; see section [10.3.1, *Generating a client secret*](#).
- `<user account>` is the logon name of the user account you created for the API; see section [10.2, *Creating a user account*](#).

10.3.3 Refreshing the authentication server settings

Once you have made your changes to the web.oauth2 server settings, you must refresh the application pool to ensure that all the systems are using the latest settings.

To refresh the server settings:

1. Recycle the web service app pool:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web service has picked up the changes to the configuration file.

2. Check that the web.oauth2 server is still operational by logging on to the MyID Operator Client.

Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

```
HTTP Error 500.30 - ANCM In-Process Start Failure
```

10.3.4 Encrypting the client secret

Because you are going to store the client secret in a configuration text file, you must encrypt it for security purposes. You can use DPAPI to encrypt the client secret; this uses the logged-on Windows user to encrypt the secret, and only the same Windows user can decrypt the secret.

To encrypt the client secret:

1. On the PC on which you intend to run the scripts, log on as the user under which you are going to run the scripts.

Note: It is important that you use this account on this PC to encrypt the secret, as no other accounts can decrypt the secret to use it.

2. Copy the `DPAPIEncrypt.ps1` script from the MyID web server to the PC on which you intend to run the scripts.

By default, this is in the following folder on the MyID web server:

```
C:\Program Files\Intercede\MyID\web.oauth2\
```

3. Open a Windows PowerShell command prompt, and navigate to the folder to which you copied the PowerShell script.
4. Run the following PowerShell script:

```
.\DPAPIEncrypt.ps1 <secret>
```

where:

- `<secret>` is the client secret that you created using the `GenClientSecret.ps1` script; see section [10.3.1, Generating a client secret](#).

For example:

```
.\DPAPIEncrypt.ps1 b5989015-bb9e-4533-874b-2b4a6a8280ed
```

The script outputs an encrypted copy of the secret; for example:

```
PS C:\Projects\Python Core API> .\DPAPIEncrypt.ps1 b5989015-bb9e-4533-874b-2b4a6a8280ed
```

```
AQAAANCMnd8BFdERjHoAwE/C [...] JwWwaKXWoS3i+ulxtmjVQyudpQ==
```

(Encrypted output string truncated for documentation purposes.)

5. Copy the encrypted secret.

You need the encrypted secret for your script settings file; see section [10.4, Creating the settings file](#).

10.4 Creating the settings file

You are going to create multiple scripts in this worked example, so it makes sense to store all of the configuration options in a settings file so that you can change the server, for example, in one place, and have the change picked up by all of the scripts.

To create the settings file:

1. On the PC on which you intend to run the scripts, create a new folder.
2. Inside this folder, create a text file called `settings.py` with the following content:

```
# Set the server, client ID, secret, and scope
server = "<server name>"
client_id = "<client name>"
encrypted_secret = "<DPAPI-encrypted client secret>"
scope = "<scope>"
```

Where:

- `<server name>` – the name of your MyID server.

For example:

`myid.example.com`

- `<client name>` – the name of the client you added to the web.oauth2 custom client configuration file.

In this example, you are using `myid.mysystem` – see section [10.3.2, Adding the API client](#).

- `<DPAPI-encrypted client secret>` – the encrypted secret you created with the `DPAPIEncrypt.ps1` script.

See section [10.3.4, Encrypting the client secret](#).

- `<scope>` – set to `myid.rest.basic`

For example:

```
# Set the server, client ID, secret, and scope
server = "myid.example.com"
client_id = "myid.mysystem"
encrypted_secret = "AQAAANCMnd8BFdERjHoAwE/C [...]
JwWwKXWoS3i+u1xtmjVQyudpQ=="
scope = "myid.rest.basic"
```

10.5 Getting an access token

Now that you have configured your server to allow access to the web.oauth2 web service, you can use this service to obtain an access token that allows you to call the MyID Core API.

10.5.1 Creating a function to obtain an access token

Create a new file in the same folder as `settings.py` and name it:

`access_token.py`

You are going to create a function in this file that calls the `/connect/token` endpoint and returns the access token:

```
import requests
import json

def getAccessToken(server, client_id, client_secret, scope):
    # Set up the header
    h = {'Content-Type': 'application/x-www-form-urlencoded'}

    # Set up the body of the request
    d = {'grant_type': 'client_credentials',
        'scope': scope,
        'client_id': client_id,
        'client_secret': client_secret}

    # Fetch the access token
    response = requests.post("https://" + server +
        "/web.oauth2/connect/token", headers=h, data=d)

    # Get the content of the response, then convert from JSON
    token = json.loads(response.content)

    # Return the token
    return token
```

How does this function work?

You pass in the server name, the client ID, the client secret, and the scope. It then calls the `/connect/token` endpoint, passing this information, and returns the response from the server in JSON format.

See section [3.3, Obtaining a server-to-server access token](#) for more information on this process.

The Python script uses the `requests` library to post to the endpoint, and the `json` library to convert the results from JSON to a Python dictionary for easier processing later. Other languages will have similar methods.

10.5.2 Creating the request script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`01-request.py`

In this file, import the settings file and module containing the access token function:

```
import settings
import access_token
```

Now you can obtain the values of the variables from the settings file:

```
# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
scope = settings.scope
```

But wait – before you can use the encrypted secret, you must first decrypt it.

10.5.3 Decrypting the secret

You are going to create a module that contains a function that decrypts the encrypted secret.

Create a new file in the same folder as `settings.py` and name it:

`DPAPI.py`

Add a function to this file that decrypts DPAPI-encrypted text.

```
import win32crypt
import base64

def decrypt(base64_string):
    # Takes a Base64-encoded string and decrypts it
    encrypted_string = base64.b64decode(base64_string)
    decrypted_string = win32crypt.CryptUnprotectData(encrypted_string, None, None, None, 0)
    return decrypted_string[1].decode("utf-8")
```

This function calls the `CryptUnprotectData` function in the Windows API to decrypt the encrypted Base64 string. Other languages will have their own implementation of the Windows API.

10.5.4 Adding the decryption call to the request script

In the `01-request.py` file, import the new DPAPI module you just created.

```
import settings
import access_token
import DPAPI
```

Now call the decrypt function from the module on the encrypted secret from the settings file:

```
# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
scope = settings.scope
client_secret = DPAPI.decrypt(encrypted_secret)
```

You now have all the information you need to obtain an access token.

10.5.5 Obtaining an access token

In the `01-request.py` file, call the `getAccessToken` function you created:

```
# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)

print("\nYou now have an access token.")
print(token)
```

You can now run the script.

If this script runs correctly, it displays the content of the response from the authentication server; for example:

```
You now have an access token.  
{  
  'access_token':  
    'eyJhbGciOiJIUzI1NiIsImtpZCI6IlRWTktYT09wVXlRPStSInR5cCI6ImF0K2p3dCJ9.eyJ1YmYiOiJlMjQ0MTExNjc5ImV4Ci6MTcwNDgxNDNC2NywiaXNzIjoiaHR0cHM6Ly9yZWZjdC5kb21haw4zNi5sb2NhbmC93ZWlub2FldGgyIiwiaXYXkiOiJoibXlpZC5yZXN0IiwiaWY2xpZnW5OXM2lkIjoibXlpZC5yZXBvcnRzIiwibXlpZFNlc3Npb25JCi6iOi03MTI0NDIyOTksN0U2NzQ0QzktnKy40S00MEY0LTg1NjEtODVBRTQ2Nm4REJFIiwianRpIjoiotVGNDg3QjJDQTNRFRDYWN0E4RkI2NzQ2MzgYqTdeRDQiLCJpYXQ1OjE3MDQ0MTExNjc5InInjb3B1IjpbIm15aWQuemVmZdC5iYXNPYyJdfDfQ.JIOhg_bx_9W2LIxB4xwh0B_NH_-jhmlBnjkoMd_ZypzHT8s10jWVq-h47ub1huqv9Yr_mq6CtO5rIRyk6UGuk635NVyPUHft2PjF0AWmAal58Rbj0cBjI7A_xeCP_26K-C06Uo44cklS7picS95hCiTwv99W9NpyKZYbUTSp-TGiVgeGoQFP5b9mb20_rPSSAM3UPZtSBSDhHzL2562g9nnarEbZCFd22h-OIYZ93sHE2cvVi_oIfHZi9yJLERTrhoMM8LuD0vk33uPoFLFrJNgZ-YjbQEVTu8Jhwt0mCR1YJvpNC89DI4M-rRI350DPugrLOAZ2TPkrN1pu7Wva',  
  'expires_in': 3600,  
  'token_type': 'Bearer',  
  'scope':  
    'myid.rest.basic'}
```

10.5.6 Troubleshooting

If you see an error similar to:

```
-2146893813, 'CryptUnprotectData', 'Key not valid for use in specified state.'
```

this means that the script has been unable to decrypt the DPAPI-encrypted client secret. Make sure that you encrypted the secret on the same PC and with the same logged-on user as you are running the script to decrypt it.

If you see an error similar to:

```
{'error': 'invalid_client'}
```

this means that the information in your settings file does not fully match the information in the custom client configuration file.

This may occur because:

- The `client_id` in the settings file does not match the `ClientId` you set up in the custom client configuration file on the MyID web server.
- The client secret is not set up correctly; make sure that the `ClientSecrets:Value` option is the Base64-encoded SHA-256 hash of the client secret you generated.

10.6 Calling the API

Now that you can obtain an access token, you can use it to call the API.

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`02-call.py`

10.6.1 Setting up the imports and variables

In this new file, import the settings file, the module containing the access token function, the DPAPI module for decrypting the client secret, and the Python requests module that you can use to send a message to the API server.:

```
import settings
import access_token
import DPAPI
import requests
```

Now you can obtain the values of the variables from the settings file and decrypt the client secret:

```
# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
scope = settings.scope
client_secret = DPAPI.decrypt(encrypted_secret)
```

10.6.2 Obtaining the access token

Call the `getAccessToken` function you created in the `access_token.py` file:

```
# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.6.3 Calling the API

Now you can call the API.

```
# Use the token to call the API
response = requests.get(
    "https://" + server + "/rest.core/api/Groups?q=*",
    headers={"Authorization": f"Bearer {token['access_token']}"},
)
print(response.text)
```

How does this code work?

It uses the requests module to send a GET request to the MyID server, with the specific endpoint; in this case, `api/Groups`, which returns a list of the groups in the MyID database

The API documentation lists all of the endpoints you can use, provides details of the parameters you can include on the URL (in this example, a simple query of `q=*` meaning "show me all of the groups") and tells you what verb to use – in this case, GET, or in other cases POST (to send data to the server) PATCH (to update data on the server) or DELETE (to delete data from the server).

It pulls the value of the `access_token` from the results returned from the call to the `web.oauth2` server, and passes it to the API in the `Authorization` header.

Note: This token is in standard JSON Web Token (JWT) format. You can inspect the token manually at jwt.io or use a variety of existing libraries to view details of the token. See also section [10.15, Working with JWTs](#).

Important: The user account you set up for the API client in the custom client configuration file for the `web.oauth2` service must have permission to access this endpoint. See section [2.2, Accessing the API features](#) for details.

For more information on calling the API, see section [5.1, Calling the API from an external system](#).

The information displayed is a block of JSON that contains details of every group returned by the query.

10.6.4 Troubleshooting

If you see an error similar to:

```
{"message": "Your current authentication level cannot access this  
information. The logon credential used, roles that logon credential can  
access and scope available to those roles may limit your  
access", "code": "WS50000"}
```

this means that the user account you set up for the API client in the custom client configuration file for the web.oauth2 service does not have permission to access this endpoint. See section [2.2, Accessing the API features](#) for details.

You can also check the `response.status_code`:

```
print(response.status_code)
```

A response of:

403

means that the user account does not have permissions.

If you get empty results from `response.text`, check the `response.status_code`. A response of:

401

possibly means that the access token has expired. This is unlikely to happen in a simple script – an access token lasts an hour – but may occur if you carry out a significant number of API calls using the same access token.

A `response.status_code` of:

200

means that the request succeeded.

The API documentation contains lists of the status codes that may be returned for each API endpoint.

10.7 Getting a list of group IDs

You have obtained an access token and used it to call the API. Now you can start to work with the data that is returned by the API.

The API documentation describes the data that is returned for each endpoint. This may vary from system to system, and may be customized for your own implementation; for example, you may have custom fields that have been added using Project Designer. The API documentation on the server is generated directly from your system and is always up-to-date.

For example, the documentation for the `/api/Groups` endpoint says that the response contains the following:

```
"results": [  
  {  
    "name": "name",  
    "desc": "desc",  
    "ou": "ou",  
    "parentName": "parent",  
    "id": "GroupUuid",  
    "href": "string"  
  },  
]
```

If you look at the raw results from your `02-call.py` script, you can see this information for each group.

For example, the `results` node might contain information similar to the following (reformatted for readability):

```
"results":[
  {
    "name":"Root",
    "desc":"Parent of all groups",
    "ou":"",
    "parentName":"",
    "id":"EA1856E3-9C8B-45FA-8209-9D006CB43863",
    "href":"groups/EA1856E3-9C8B-45FA-8209-9D006CB43863"
  },
  {
    "name":"MasterAdmin",
    "desc":"System Administrators",
    "ou":"",
    "parentName":"Root",
    "id":"483040F1-52FE-4CE1-ADCF-3FC67065003F",
    "href":"groups/483040F1-52FE-4CE1-ADCF-3FC67065003F"
  },
  {
    "name":"System Startup",
    "desc":"System generated group for bootstrapping the process",
    "ou":"",
    "parentName":"Root",
    "id":"DD87FFA6-881E-4737-92D1-3DF862CF3CF5",
    "href":"groups/DD87FFA6-881E-4737-92D1-3DF862CF3CF5"
  },
  {
    "name":"Enterprise",
    "desc":"",
    "ou":"OU=Enterprise,DC=domain31,DC=local",
    "parentName":"Default ADS",
    "id":"B2F202C3-C1FD-48ED-A955-20E1B019907A",
    "href":"groups/B2F202C3-C1FD-48ED-A955-20E1B019907A"
  },
  {
    "name":"Finance",
    "desc":"",
    "ou":"OU=Finance,OU=Enterprise,DC=domain36,DC=local",
    "parentName":"Enterprise",
    "id":"EF8998AC-B2B6-4CF0-8DC3-99D0DCA2DA2B",
    "href":"groups/EF8998AC-B2B6-4CF0-8DC3-99D0DCA2DA2B"
  }
]
```

In this example, you are going to pull out the names and IDs of the groups.

10.7.1 Getting started

Copy your `02-call.py` script and rename it:

`03-groups.py`

Add the `json` module to the imports. The response is in JSON format, and the `json` module lets you work with JSON formatted data easily. Other languages have their own ways of working with JSON.

```
import access_token
import DPAPI
import settings
import requests
import json
```

You do not need to change any of the settings, or the access token section, and you can keep the call to the `/api/Groups` endpoint the same as the previous script. The information you receive will be the same, and the only difference will be that you process it before displaying the results.

Delete the last line in the script:

```
print(response.text)
```

You do not want to display the raw data.

10.7.2 Processing the results

Add the following to your script:

```
# Extract the list of group IDs returned by the API call
print("\nThe following is the list of groups and their IDs:")
data = json.loads(response.text)
for group in data["results"]:
    print(group["name"] + ": " + group["id"])
```

How does this code work?

The script loads the JSON from the `response.text` and turns it into a Python dictionary. This lets you access each part of the JSON.

For each item in the `results` node of the JSON, the script prints the `name` and `id` values. For example:

```
The following is the list of groups and their IDs:
Root: EA1856E3-9C8B-45FA-8209-9D006CB43863
MasterAdmin: 483040F1-52FE-4CE1-ADCF-3FC67065003F
System Startup: DD87FFA6-881E-4737-92D1-3DF862CF3CF5
Enterprise: B2F202C3-C1FD-48ED-A955-20E1B019907A
Finance: EF8998AC-B2B6-4CF0-8DC3-99D0DCA2DA2B
```

IDs are essential to working with the MyID Core API. Whenever you want to refer to a group, or a person, or a device, and so on, you must know the appropriate ID. These IDs are unique to your installation of MyID; even if you have a Finance group on your system that is otherwise identical, the ID will not be `EF8998AC-B2B6-4CF0-8DC3-99D0DCA2DA2B`.

Important: You are going to need a group ID for the next example script. Find and take a note of the ID of a group into which you are going to add a person.

10.7.3 Troubleshooting

If you see an error similar to:

```
NameError: name 'json' is not defined. Did you forget to import 'json'?
```

make sure that you included the `json` module in the list of imports at the top of the file.

10.8 Adding a person

The MyID Core API does not just let you obtain information from the MyID database (such as the list of group IDs from the previous example script) – it also allows you to update the MyID database.

The example script in this section adds a new person to your MyID system using a `POST` HTTP method to the `api/People` endpoint.

10.8.1 Setting up the import data

The MyID Core API uses JSON for the payload for your calls to the web service. To make things easy, you are going to create a JSON file that contains the details for the person you want to add.

Create a new text file in the same folder as `settings.py` and `access_token.py` and name it:

`addperson.json`

Add the following to the new file:

```
{
  "account": {
    "cn": "Susan Smith",
    "dn": "cn=Susan Smith, o=example, c=us",
    "domain": "example",
    "ou": "ou=example, c=us",
    "samAccountName": "Susan Smith",
    "sysLogonName": "Susan Smith",
    "upn": "susan.smith@example.com"
  },
  "contact": {
    "emailAddress": "susan.smith@example.com"
  },
  "employeeId": "11223344",
  "enabled": "1",
  "name": {
    "first": "Susan",
    "last": "Smith",
    "title": "Ms"
  },
  "group": {
    "id": "EF8998AC-B2B6-4CF0-8DC3-99D0DCA2DA2B"
  },
  "logonName": "susan.smith",
  "roles": [
    {
      "id": "Cardholder",
      "scope": "self"
    }
  ],
  "personal": {
    "dateOfBirth": "1970-01-01"
  }
}
```


You must amend the following details:

- `group:id` – set this to the ID of a group in your system. Use one of the IDs that you discovered from the previous example script; see section [10.7, *Getting a list of group IDs*](#).
- `logonName` – make sure this is a unique logon name that has not been used for anyone else in your MyID system. You can search for existing logon names using the **Logon** field in the **People** category in the MyID Operator Client.
- `roles` – if you do not have a `Cardholder` role, set the `id` to the name of a role that exists in your system.

Note: There are far more attributes available for people in MyID than are listed in this sample file, and the API documentation provides details of the structure of the JSON you need to provide those attributes. The above sample import file is a minimal set of attributes just for proof of concept. You may also have custom attributes that have been added to your system by Project Designer – the MyID Core API allows you to populate those custom attributes, and the API documentation provides details of where they fit in the JSON structure.

10.8.2 Creating your add person script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`04-add-person.py`

Add the usual import, settings, and access token code:

```
import access_token
import DPAPI
import settings
import requests
import json

# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.8.3 Loading the person's data

Add the following to the script:

```
# Load the person data from the JSON file
with open('addperson.json') as f:
    addperson = f.read()
```

This opens the `addperson.json` file you created and reads its contents.

10.8.4 Calling the API to add a person

Add the following to the script:

```
# Call the API to add the person to MyID
response = requests.post(
    "https://" + server + "/rest.core/api/People?confirm=false",
    headers={"Authorization": f"Bearer {token['access_token']}",
            "Content-Type": "application/json-patch+json",
            "accept": "application/json"},
    data=addperson)
```

How does this script work?

It sends a `POST` request to the `/api/People` endpoint on the server with the appropriate headers and includes the JSON data from the imported file in its body.

You can find the available parameters, headers, and content format for each endpoint in the API documentation.

10.8.5 Checking the response

Add the following to the script:

```
# Check the response status code
if response.status_code==200:
    print("Person added to MyID with an ID of:")
    returnedData = json.loads(response.text)
    print(returnedData["id"])
else:
    print("An error occurred:")
    print(response.text)
```

This checks that the import succeeded (a `status_code` of 200) and displays the ID of the new person.

For example:

```
Person added to MyID with an ID of:
EF84D588-8C0F-4476-BC85-C4A896970E44
```

If the import did not succeed, the script displays the error message.

10.8.6 Troubleshooting

If you see an error similar to:

```
{"message": "Validation problem, the value for 'logonName', is mandatory", "code": "WS30015", "invalidField": "logonName", "invalidFieldReason": "is mandatory"}
```

this means that the `addperson.json` file was not formatted correctly and did not contain a `logonName` field.

If you see an error similar to:

```
{"message": "Validation problem, the value for 'roles', invalid role specified", "code": "WS30004", "invalidField": "roles", "invalidFieldReason": "invalid role specified"}
```

this means that the role you specified in the import file is not correct. Make sure you have specified the name of an existing role in your MyID system. This error may also occur if you specified an incorrect value for the `scope` – make sure the scope is one of the following:

- `self`
- `department`
- `division`
- `all`

If you see an error similar to:

```
No such file or directory: 'addperson.json'
```

This means that either you have not put your `addperson.json` in the correct location, or you are experiencing a path issue with how your Python file is being run. If you are experiencing a path issue, either add a specific path to the location of the `addperson.json` file (for example `C:\TestFolder\addperson.json`) or run your Python file from the same folder as the example script is located.

If you see an error similar to:

```
{"message": "The item referenced was not found", "code": "WS40005"}
```

this may mean that the ID you specified for the group was not valid. Make sure that you have used the ID of an existing group in the MyID database.

If you see an error similar to:

```
{"message": "Validation problem, the value for 'name.first', 'First Name' or 'Last Name' must be provided", "code": "WS30017", "invalidField": "name.first", "invalidFieldReason": "'First Name' or 'Last Name' must be provided"}
```

this means that you have not provided sufficient information about the person's name.

You may see other, similar, messages if you have failed to provide the minimum required information about a person, or have not provided an accepted value for a particular field.

Look up the error code (for example, `WS30017`) in the *MyID Operator Client error codes* section of the [Error Code Reference](#) guide.

If you experience any other issues, particularly on a system that has been customized, it is worth checking in the API documentation that the structure of the JSON for a person matches the structure in the example; in some cases you may have to rewrite the `addperson.json` import file to match the structure for your customized system.

10.9 Updating a person

The `POST /api/People` endpoint (as used in the previous example script) allows you to *add* a person to your MyID system; the `PATCH /api/People` endpoint allows you to *update* an existing person.

In this example, you are going to update the person you added in the previous example script and change their last name.

10.9.1 Creating your update person script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`05-update-person.py`

Add the usual import, settings, and access token code:

```
import access_token
import DPAPI
import settings
import requests
import json

# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.9.2 Setting up the update data

Instead of using an external file, for this example you can create the patch body in the script:

```
# Create JSON for the patch body
surname = "Jones"
updateData = {
    "name": {
        "last": surname
    }
}
updatePerson = json.dumps(updateData)
```

You could use this method to create update data from elsewhere; for example, you could pull the person's updated surname from a different system and create the required JSON from this information.

Note: When you update a person, you do not have to provide the entire schema of attributes for the person, but just the attribute you want to change.

10.9.3 Finding the ID of the person

To make any changes to a person, you must know their unique ID. You can retain this from the response returned when you added the person to your MyID system, or you can look it up. In this example, you are going to use the person's logon name to look up their ID.

Add the following to the script:

```
# Specify the person we want to update. We know the logon name but not the ID.
personLogon = "susan.smith"

# Call the API to get the person's details
response = requests.get(
    "https://" + server + "/rest.core/api/People?logonName=" + personLogon,
    headers={"Authorization": f"Bearer {token['access_token']}"},
)
returnedData = json.loads(response.text)
```

How does this script work?

It uses a GET call to the `/api/People` endpoint with the person's logon name as a parameter. This returns a block of data containing all the information about the matching person; you will use this to obtain the person's unique ID.

10.9.4 Calling the API to update a person

Add the following to the script:

```
# Extract the person's ID from the person's data
if len(returnedData["results"]) > 0:
    personID = returnedData["results"][0]["id"]

# Call the API to update the person in MyID
response = requests.patch(
    "https://" + server + "/rest.core/api/People/" + personID + "?confirm=false",
    headers={"Authorization": f"Bearer {token['access_token']}",
            "Content-Type": "application/json-patch+json",
            "accept": "application/json"},
    data=updatePerson)

# Check the response status code
if response.status_code==200:
    print("Person updated in MyID. The new last name is:")
    returnedData = json.loads(response.text)
    print(returnedData["name"]["last"])
else:
    print("An error occurred:")
    print(response.text)
    print(response.status_code)
else:
    print("Error: a person with the logon name " + personLogon + " was not found.")
```

This section of the code first obtains the ID of the person.

The person's ID is returned as the `id` parameter of the first (and only) entry in the results. (There is a maximum of one entry in the results because the search query uses the logon name, which is unique.) If there are no results returned, this means that there was no match for the logon name you specified, and the script displays a message to that effect.

The script then calls the API to pass the updated name data, using the person's ID to specify which person to update.

Finally, the script then checks the response to ensure that the API call succeeded (a `status_code` of 200) and displays the name that was returned in the response for confirmation:

```
Person updated in MyID. The new last name is:
Jones
```

10.9.5 Troubleshooting

If you see a message similar to:

```
Error: a person with the logon name susan.smith was not found.
```

this means that the logon name you specified in the `personLogon` variable could not be matched against the MyID database.

If you see a message similar to:

```
Person updated in MyID. The new last name is:  
Smith
```

where the new name does not match the name you specified in this script, check that you have provided the `name` and `last` data correctly; if you provide attributes that do not exist (for example, by introducing a typo such as `lost`) there is no error displayed, and the `status_code` is 200. This is why it is important to verify that the data returned by the call to the API is as expected.

10.10 Uploading a picture

You can use the `PATCH /api/People` endpoint to update a person's attributes; however, for special cases such as image fields, the API provides specific endpoints that allow you to upload data to the server. In this example, you are going to use the following endpoint to upload a user image for the person you added and updated in the previous example scripts:

```
POST /api/People/{id}/images/photo
```

10.10.1 Providing an image

Find a JPEG image on your PC, copy it into the same folder as `settings.py` and `access_token.py`, and rename it:

```
susan.smith.jpg
```

10.10.2 Creating your upload image script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

```
06-upload-image.py
```

To upload an image through the API, you need to convert it to Base64, so add the Python `base64` module to your list of imports:

```
import access_token
import DPAPI
import settings
import requests
import json
import base64
```

You can now add the usual settings and access token code:

```
# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.10.3 Loading the image from a file

Add the following to the script:

```
# Load the image from file and convert it to Base64
with open("susan.smith.jpg", "rb") as f:
    img_base64 = base64.b64encode(f.read()).decode('utf-8')
```

This loads the binary data from the file then converts it to Base64.

Whenever you want to upload binary files (for example, images) to MyID, you must first convert them to Base64 so that you can include them in the JSON data that you post to the API endpoint.

10.10.4 Setting up the update data

Add the following to the script:

```
# Create JSON for the request body
updateData = {
    "image": {
        "data": img_base64,
        "mimetype": "image/jpeg"
    }
}
updatePerson = json.dumps(updateData)
```

This takes the Base64-encoded copy of the image and adds it to the appropriate structure. You must also provide the `mimetype` – in this case, `image/jpeg`.

10.10.5 Finding the ID of the person

You can use the same method to find the ID of the person from their logon name as you used in the previous script:

```
# Specify the person we want to update. We know the logon name but not the ID.
personLogon = "susan.smith"

# Call the API to get the person's details
response = requests.get(
    "https://" + server + "/rest.core/api/People?logonName=" + personLogon,
    headers={"Authorization": f"Bearer {token['access_token']}"},
)
returnedData = json.loads(response.text)
```

10.10.6 Calling the API to upload a picture

Add the following to the script:

```
# Extract the person's ID from the person's data
if len(returnedData["results"]) > 0:
    personID = returnedData["results"][0]["id"]

# Call the API to update the person in MyID
response = requests.post(
    "https://" + server + "/rest.core/api/People/" + personID + "/images/photo",
    headers={"Authorization": f"Bearer {token['access_token']}",
            "Content-Type": "application/json-patch+json",
            "accept": "image/jpeg"},
    data=updatePerson)

# Check the response status code
if response.status_code==200:
    print("Person's picture updated in MyID.")
else:
    print("An error occurred:")
    print(response.text)
else:
    print("Error: a person with the logon name " + personLogon + " was not found.")
```

This section of the code first obtains the ID of the person, just as you did with the previous example.

The person's ID is returned as the `id` parameter of the first (and only) entry in the results. If there are no results returned, this means that there was no match for the logon name you specified, and the script displays a message to that effect.

The script then calls the API to upload the image, using the person's ID to specify the person to whom the image belongs.

Finally, the script then checks the response to ensure that the API call succeeded (a `status_code` of 200) or displays an error.

The call to the API returns the image that was uploaded; you can compare this to the image you sent to confirm that there were no problems, if you want:

```
returned_img_base64 = base64.b64encode(response.content).decode('utf-8')
if (returned_img_base64 == img_base64):
    print("Uploaded and downloaded images match.")
else:
    print("A problem has occurred. Uploaded and downloaded images do not match.")
```

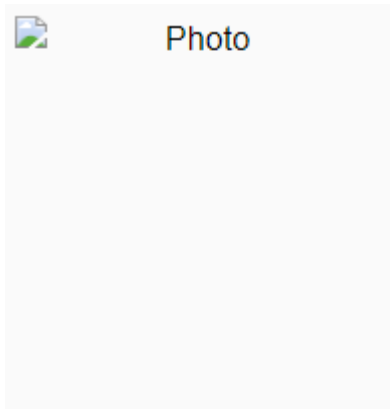
10.10.7 Troubleshooting

If you see an error similar to:

```
FileNotFoundError: [Errno 2] No such file or directory: 'susan.smith.jpg'
```

this means that the script could not load the `susan.smith.jpg` image file. Make sure it is in the same folder as the scripts.

If the image looks similar to the following in the MyID Operator Client:



this means the image file type was not recognized. Make sure your image file is a JPG, PNG, or GIF.

If you see an error similar to:

```
{"type": "https://tools.ietf.org/html/rfc9110#section-15.5.1", "title": "Bad Request", "status": 400, "traceId": "00-79ae2ed2dfecdd0e26de9a8beac46b46-0ff112c56ab2c38c-00"}
```

this may have been caused by a zero-length image file.

If you see an error similar to:

```
jpg not found error
```

This means that either you have not put your `susan.smith.jpg` in the correct location, or you are experiencing a path issue with how your Python file is being run. If you are experiencing a path issue, either add a specific path to the location of the `susan.smith.jpg` file (for example, `C:\TestFolder\susan.smith.jpg`) or run your Python file from the same folder as the example script is located.

10.11 Getting credential profiles

One important feature of MyID is the ability to create requests for devices for the people in your system. For example, you may want to request a smart card for your newly-added person.

When you request a device, you specify a credential profile – this describes the type of device, the issuance process, the certificates, the printed layout, and which people (operators and end users) can receive, request, validate, or collect the requests.

In the MyID Operator Client, when you are presented with a list of credential profiles to choose from, MyID presents only those credential profiles that the recipient is permitted to receive, and which you as an operator are permitted to request. The API provides an endpoint that provides the same list of credential profiles:

```
GET /api/People/{id}/credProfiles
```

10.11.1 Creating your update person script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

```
07-credential-profiles.py
```

Add the usual import, settings, and access token code:

```
import access_token
import DPAPI
import settings
import requests
import json

# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.11.2 Finding the ID of the person

You can use the same method to find the ID of the person from their login name as you used in the previous scripts:

```
# Specify the person we want to update. We know the login name but not the ID.
personLogin = "susan.smith"

# Call the API to get the person's details
response = requests.get(
    "https://" + server + "/rest.core/api/People?loginName=" + personLogin,
    headers={"Authorization": f"Bearer {token['access_token']}"},
)
returnedData = json.loads(response.text)
```

10.11.3 Calling the API to get a list of credential profiles

Add the following to the script:

```
# Extract the person's ID from the person's data
if len(returnedData["results"]) > 0:
    personID = returnedData["results"][0]["id"]

    # Call the API to get the list of credential profiles
    response = requests.get(
        "https://" + server + "/rest.core/api/People/" + personID +
        "/credProfiles?canReceive=true&canRequest=me",
        headers={"Authorization": f"Bearer {token['access_token']}"})
    returnedData = json.loads(response.text)

    # Check the response status code
    if response.status_code==200:
        # Print the header of the table
        print("ID" + " " * 35 + "Name")
        # Print the body of the table
        for result in returnedData["results"]:
            print(result["id"] + " " + result["name"])
    else:
        print("An error occurred:")
        print(response.text)
        print(response.status_code)
else:
    print("Error: a person with the logon name " + personLogon + " was not found.")
```

This section of the code first obtains the ID of the person, just as you did with the previous examples.

The person's ID is returned as the `id` parameter of the first (and only) entry in the results. If there are no results returned, this means that there was no match for the logon name you specified, and the script displays a message to that effect.

The script then calls the API to get a list of credential profiles for that person, using the person's ID to specify the person for whom you want to request a device, setting the `canReceive` parameter to `true` to return only those credential profiles that the person can receive, and setting the `canRequest` parameter to `me` to return only those credential profiles that the logged-on user (in this case, the API client user) can request. If you do not set the `canReceive` or `canRequest` parameters, the API returns a list of *all* credential profiles, whether or not the person can receive them or the client API user can request them.

Finally, the script then checks the response to ensure that the API call succeeded (a `status_code` of 200) or displays an error. If the response is correct, the script displays a table of the credential profiles. The call to the API returns a block of JSON from which you can pull out the information you need; in this case, the script prints a simple table of the IDs and names of the credential profiles.

Take a note of the ID of one of the credential profiles that the person can receive; you will need this for the next example, where you request a device for this person.

10.11.4 Troubleshooting

If the API returns a list of all credential profiles rather than just the credential profiles the person can receive and the API user can request, check that you included the `canReceive` and `canRequest` parameters in the call to the API.

If the API returns no credential profiles, check that the person has roles that allow them to receive credential profiles. You can check this by logging on to the MyID Operator Client with the API user you created, searching for the person you added, and attempting to request a device. The list of credential profiles that appears on the Request Device Issuance screen should be the same as the list returned by your script.

10.12 Requesting a device

Now you know which credential profiles are available for the person, you can request a device.

10.12.1 Creating your request device script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`08-request-device.py`

Add the usual import, settings, and access token code, as well as adding the `datetime` module:

```
import access_token
import DPAPI
import settings
import requests
import json
from datetime import date

# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.12.2 Finding the ID of the person

You can use the same method to find the ID of the person from their logon name as you used in the previous scripts:

```
# Specify the person we want to update. We know the logon name but not the ID.
personLogon = "susan.smith"

# Call the API to get the person's details
response = requests.get(
    "https://" + server + "/rest.core/api/People?logonName=" + personLogon,
    headers={"Authorization": f"Bearer {token['access_token']}"},
)
returnedData = json.loads(response.text)
```


10.12.3 Setting the ID of the credential profile

You need the ID of a credential profile. From the list that was returned by the previous example (see section [10.11, *Getting credential profiles*](#)) select the credential profile you want to use, and add its ID to your script. You can add a job label to the request.

```
credentialProfile = "177732B8-ED11-497A-AFA3-73A8C0E79964"
requestData = {
  "credProfile": {
    "id": credentialProfile
  },
  "jobLabel": "Core API request " + date.today().strftime("%B %d, %Y")
}
deviceRequest = json.dumps(requestData)
```

Note: If you have the **Set expiry date at request** option set (on the **Issuance Processes** tab of the **Operation Settings** workflow) you can also add a specific expiry date to the request JSON:

```
"expiryDate": "2024-12-25"
```

10.12.4 Calling the API to request a device

Add the following to the script:

```
# Extract the person's ID from the person's data
if len(returnedData["results"]) > 0:
    personID = returnedData["results"][0]["id"]

# Call the API to request the device
response = requests.post(
    "https://" + server + "/rest.core/api/people/" + personID + "/requests",
    headers={"Authorization": f"Bearer {token['access_token']}",
             "Content-Type": "application/json-patch+json",
             "accept": "application/json"},
    data=deviceRequest)

# Check the response status code
if response.status_code==200:
    print("Request added to MyID with an ID of:")
    returnedData = json.loads(response.text)
    print(returnedData["id"])
else:
    print("An error occurred:")
    print(response.text)
    print(response.status_code)
else:
    print("Error: a person with the logon name " + personLogon + " was not found.")
```

This section of the code first obtains the ID of the person, just as you did with the previous examples.

The person's ID is returned as the `id` parameter of the first (and only) entry in the results. If there are no results returned, this means that there was no match for the logon name you specified, and the script displays a message to that effect.

The script then calls the API to create a device request for the person. The details of the request are included in the body of the POST to the API.

Finally, the script then checks the response to ensure that the API call succeeded (a `status_code` of 200) and displays the ID of the newly-created request. Take a note of the request ID, as you will need it for the next example, where you check the status of the request.

10.12.5 Troubleshooting

If you see an error similar to:

```
{"message":"The person selected does not have a Distinguished Name. This profile requires a Distinguished Name for credential issuance.", "code":"WS50017"}
```

check that the person for whom you are requesting a device has a Distinguished Name configured – you can check this on the **Account** tab of the View Person screen in the MyID Operator Client.

If you see an error similar to:

```
{"message":"Your assigned roles do not have permission to request the credential profile specified", "code":"WS50008"}
```

check that the MyID user specified in the client settings for you system (in this example, `api.external`) has permission to request the credential profile.

If you see an error similar to:

```
{"message":"The person selected does not have a role assigned that can hold the requested credential profile", "code":"WS50011"}
```

check that the person for whom you are requesting the device has permissions to receive the credential profile.

If you see an error similar to:

```
{"message":"The person selected does not have facial biometrics. The credential profile requires that the recipient has facial biometrics enrolled.", "code":"WS50034"}
```

check that the credential profile has not been configured to require facial biometrics.

If you see an error similar to:

```
{"message":"The person selected does not have all required information for this credential profile. Check the Person History audit details to identify the missing requisite user data", "code":"WS50016"}
```

check that the credential profile has not been configured for Requisite User Data that the person does not have.

10.13 Checking the status of a request

Now that you have requested a device for a person, you can check the status.

You can obtain the ID of a request in a variety of ways; for example:

- Calling `GET /api/People/{id}/requests` to get a list of requests for a person.
- Calling `GET /api/Reports/100406` to run the **Requests** report.
- Calling `GET /api/Requests` to obtain a list of requests.
- Calling `GET /api/Devices/{id}/requests` to obtain a list of requests for a particular device.
- Retaining the ID returned when you request a device.

In this example, for simplicity, you are going to use the ID of the request returned in the previous example.

10.13.1 Creating your request status script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`09-request-status.py`

Add the usual import, settings, and access token code:

```
import access_token
import DPAPI
import settings
import requests
import json

# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.13.2 Setting the ID of the request

You need the ID of a request. Find the ID that was returned by the previous example (see section [10.12, Requesting a device](#)) and add it to your script.

```
requestID = "05AE83AB-0831-44C4-A718-65AD70E6592E"
```

10.13.3 Calling the API to get the request status

Add the following to your script:

```
# Use the token to call the API
response = requests.get(
    "https://" + server + "/rest.core/api/requests/" + requestID,
    headers={"Authorization": f"Bearer {token['access_token']}"},
)
returnedData = json.loads(response.text)
```

How does this script work?

It sends a GET request to the `/api/requests` endpoint on the server, specifying the request ID. This returns information about the request.

10.13.4 Creating your notification routines

When you get the status of the request, you are going to want to carry out different actions depending on the status. For example, if the request is awaiting issuance, you may want to send a reminder to the cardholder; if the request is awaiting validation, you may want to update a list of requests for operators to approve; if the request has completed, you may want to display details about the issued device.

In this sample code there is no information sent; the information is printed out into the terminal. If you want to set up a Python system that sends email messages, you can consider using the `smtplib` library.

Add the following to your script:

```
def notify_operator(request):
    print("A request has been made for", request["target"]["name"],
          "to have a device issued with the credential profile of",
          request["credProfile"]["name"],
          "at", request["initiationDate"] + ".")
    print("This request is awaiting validation.")

def issued_device(request):
    print("The request has been issued to the following device:",
          request["device"]["dt"],
          "serial number:",
          request["device"]["sn"])

def notify_cardholder(request):
    print("A request has been made for", request["target"]["name"],
          "to have a device issued with the credential profile of",
          request["credProfile"]["name"],
          "at", request["initiationDate"] + ".")
    print("This request is awaiting issuance.")
```

10.13.5 Checking the status of the request

Now you can check the status of the request, and carry out the appropriate actions:

```
try:
    match returnedData["status"]:
        case "Awaiting Validation":
            notify_operator(returnedData)
        case "Completed":
            issued_device(returnedData)
        case "Awaiting Issue":
            notify_cardholder(returnedData)
        case "Cancelled":
            print("The request has been cancelled. No further action required.")
        case _:
            print("The request has status " + returnedData["status"] + ".")
except:
    print("An error occurred:")
    print(returnedData)
```

10.13.6 Troubleshooting

If an error similar to the following appears:

```
{'message': 'Invalid data supplied', 'code': 'WS30001'}
```

check that the request ID is a valid GUID.

If an error similar to the following appears:

```
{'message': 'The item referenced was not found', 'code': 'WS40005'}
```

check that the request ID exists.

10.14 Getting multiple pages of reports

One feature of the MyID Core API that you need to be aware of is report paging. If there is a lot of data to be returned from a report, the MyID Core API delivers it page by page.

You can see this in action in the MyID Operator Client, where extra pages of the report are loaded as you scroll down the results.

The JSON output from a report provides a link you can use to obtain the next page of results, if necessary; for example:

```
{
  "op": "nextPage",
  "cat": "page",
  "desc": "Next Page of Results",
  "verb": "GET",
  "href": "reports/290015?page=2&order_by=-UTCStartDateTime"
}
```

You can also keep calling the API until the page you requested returns an empty set of results, as in the following example.

10.14.1 Creating your report paging script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`10-report-paging.py`

Add the usual import, settings, and access token code:

```
import access_token
import DPAPI
import settings
import requests
import json

# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.14.2 Creating a function to return a specific page of a report

Add the following to your script:

```
# Create a function to request a specific page of a report
def getReportPage(server, report, page, token):
    response = requests.get(
        "https://" + server + "/rest.core/api/reports/" + report + "?page=" + str(page),
        headers={"Authorization": f"Bearer {token['access_token']}"},
    )
    return response
```

This code creates a function into which you pass the server, report ID, the page you want to retrieve, and your access token. It then returns the response for a specific page of the report.

10.14.3 Setting up the variables

Add the following to your script:

```
# Set up the variables for the report request
page = 0
report = "290015"
results = [{'dummy': ''}] # Dummy results which are overwritten
allResults = []
```

Report ID 290015 is the Unrestricted Audit Report, which is ideal for this example as it usually contains a large amount of information over several pages.

10.14.4 Calling the report page function

Add the following to your script:

```
# Keep calling the report API until there are no more results returned
while results != []:
    page += 1
    response = getReportPage(server, report, page, token)
    if response.ok:
        data = json.loads(response.text)
        results = data["results"]
        allResults += results
    else:
        print("An error occurred:\n" + response.text)
        break
page -= 1
```

How does this code work?

It keeps calling the `getReportPage` function until that function returns an empty `results` node. It increments the page count each time. Each page of results is appended to the `allResults` variable.

10.14.5 Providing the results

Because there is going to be a large amount of JSON data, there is little point printing it out to the console, so instead the script displays the total number of pages returned from the report, and writes the results to a `temp.json` file.

```
# Print how many pages of report were returned
if page == 1:
    strPages = "page"
else:
    strPages = "pages"
print("A total of " + str(page) + " " + strPages + " of results.")

# If at least one page of report was returned, write the results to a file
if page > 0 :
    with open("temp.json", 'w') as f:
        f.write(json.dumps(allResults))
```

10.14.6 Troubleshooting

If you see an error similar to:

```
{"message": "Your current authentication level cannot access this
information. The logon credential used, roles that logon credential can
access and scope available to those roles may limit your
access", "code": "WS50000"}
```

this means that the user (in this example, `api.external`) does not have access to the Unrestricted Audit Report. Use the **Edit Roles** workflow to ensure that the user's role has access to this report.

10.15 Working with JWTs

When you request an access token from the web.oauth2 web service, it is returned as a JSON Web Token (JWT). This is a standard format that provides information about the token, including its expiry date.

You can inspect a token using standard tools; for example, you can use the jwt.io website.

You can also inspect a token using a variety of libraries. In this example, you can use the Python `jwt` library.

10.15.1 Creating your JWT script

Create a new file in the same folder as `settings.py` and `access_token.py` and name it:

`11-jwt.py`

Add the following imports:

```
import access_token
import DPAPI
import settings
import requests
import datetime
import jwt
from jwt import PyJWKClient
```

This example requires the `datetime` module to process the expiry date, as well as `jwt` to decode the JWT and `PyJWKClient` to verify the JWT signature. You must also have the `cryptography` module installed if you want to verify the digital signature:

```
py -m pip install pyjwt
```

```
py -m pip install cryptography
```

Add the usual settings and access code:

```
# Get the settings from the settings.py file
server = settings.server
client_id = settings.client_id
encrypted_secret = settings.encrypted_secret
client_secret = DPAPI.decrypt(encrypted_secret)
scope = settings.scope

# Get the access token
token = access_token.getAccessToken(server, client_id, client_secret, scope)
```

10.15.2 Decoding the JWT

You can now decode the access token JWT.

Add the following code:

```
# Decode the JWT
decoded = jwt.decode(token["access_token"], options={"verify_signature": False})

# Obtain the expiry time from the decoded JWT
print("\nExpiry time decoded from the JWT: ")
print(datetime.datetime.fromtimestamp(decoded["exp"]))
```

This uses the `jwt` module to decode the token without verifying its signature. It then displays the expiry date from the token.

If you want to view all the details of the decoded JWT, use the following:

```
print(decoded)
```

You can run the script now.

10.15.3 Verifying the signature

You can also decode the access token JWT while verifying the signature. This takes a few more steps to obtain the signing key.

Add the following code:

```
# Obtain the JWT signing key
response = requests.get("https://" + server + "/web.oauth2/.well-known/openid-configuration")
jwks_uri = response.json()['jwks_uri']
jwks_client = PyJWKClient(jwks_uri)
signing_key = jwks_client.get_signing_key_from_jwt(token["access_token"])

# Decode the JWT, validating the signing key
try:
    data = jwt.decode(
        token["access_token"],
        signing_key.key,
        algorithms=["RS256"],
        audience="myid.rest",
        options={"verify_exp": True},
    )
    # Obtain the expiry time from the decoded JWT
    print("\nExpiry time decoded from the JWT with signature validation: ")
    print(datetime.datetime.fromtimestamp(data["exp"]))
except Exception as e: print(e)
```

How does this code work?

First you need to find the location of the JSON Web Key Set (JWKS), which is the set of keys that you can use to verify the JWT. You can obtain this from the configuration of the web.oauth2 server:

```
http://<server>/web.oauth2/.well-known/openid-configuration
```

The script pulls the URI of the JWKS from this configuration information, then creates a Python JWK client from the URI. It then obtains the signing key from this client.

The script can then decode the JWT, like the previous section, but this time passing in the signing key. If the signing key is valid, it displays the expiry time.

10.15.4 Troubleshooting

If you see an error similar to:

```
Signature verification failed
```

this means that the script could not verify the signature. It is possible that the JWT was created from a different server. This is unlikely to happen in this script, but it is a case you must consider when verifying the signature of a JWT in your own projects.

11 Example – user authentication

This chapter contains a worked example of creating a web page that calls the MyID Core API from a web browser using client authentication. The web page offers a login button; the user clicks through to the MyID authentication server, provides their authentication (for example, types their username and security questions, or inserts their smart card and types their PIN) and is then returned to the web page, with access to all the features of MyID that their user account allows. The web page can then use this access to allow the user to view a list of people, then select an individual person to view further details about that person.

By the time you complete these examples, you should have a basic understanding of how to authenticate to the server and call the API from a web page.

Each stage of this worked example relies on you having completed the earlier stages. If you want to go through these worked examples, you must do so in the order provided.

Note: A full listing of the final stage of the example is provided at the end of this chapter; this shows all of the parts of the example code in their context, so you can use this to orient yourself if you are unsure where some of the code should go. See section [11.10, Example code listing](#).

Important: This chapter contains code samples that you can copy and paste into your own project. Due to the limitations of PDFs, some whitespace characters may not come across cleanly; this can cause issues with JSON files particularly. You are recommended to validate and sanitize your code samples before using them; for example, you can use a JSON reformatter to ensure that the JSON files have the correct format.

11.1 Requirements

The example website has been developed using JavaScript, and has been tested in Google Chrome, Microsoft Edge, and Mozilla Firefox on a Windows PC.

The website was developed using Visual Studio Code as an HTML editor:

code.visualstudio.com

To provide a local web server, this example uses the Live Server extension for Visual Studio Code:

marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer

You can use any HTML editor and web server you want. Note, however, that you *must* use a web server to host your web pages; you cannot use local HTML files running from the file system, as the MyID authentication server does not allow cross-origin resource sharing from `file://` protocols. These examples assume you are running from a local web server as provided by the Live Server extension with an address of:

`http://127.0.0.1:5500/`

Important: This example expects that you have some knowledge of JavaScript; it is not intended to teach you how to create websites. Consult a professional web developer for advice on best practices; the JavaScript samples in this chapter are intended to teach the principles of calling the MyID Core API using JavaScript, and are *not* intended for use in production environments.

11.2 Authenticating to the MyID Core API

The MyID Core API (rest.core) requires authentication through the MyID web.oauth2 service. You must configure the web.oauth2 service to add a new client to be used by your website, and configure it to allow end-user authentication.

11.2.1 Adding the API client

The sample API website uses a new client called `myid.mywebsite` – you must add this client to the configuration of the web.oauth2 service.

First, create a custom client configuration file.

1. On the MyID web server, navigate to the `CustomClients` folder.

By default, this is:

```
C:\Program Files\Intercede\MyID\web.oauth2\CustomClients\
```

If this folder does not already exist, create it.

2. In a text editor, create a `.json` file to contain your client configuration.

You can use any filename with a `.json` extension; you are recommended to use the name you have provided for the client as the filename.

You can create a custom `.json` file for each client that you want to add. You can include only one client in each file, but you can have multiple files if you need multiple clients. These clients are added to the `Clients` array from the `appsettings.json` file. You must use a unique `ClientID`; if you use the same `ClientID` in a custom file as an already existing client in the `appsettings` file, the information from the custom file completely replaces the information in the `appsettings.json` or `appsettings.Production.json` override file.

The order of precedence is:

- any `.json` file in the `CustomClients` folder.
- `appsettings.Production.json`
- `appsettings.json`

Note: The `appsettings.Production.json` file overrides the `appsettings.json` file *not* by using the client ID, but by the index of the entry in the `Clients` array. For this reason, you are recommended *not* to use the `appsettings.Production.json` file to provide the details of custom clients, but to use separate files in the `CustomClients` folder instead.

For more information about custom configuration files, see section 9, [Custom configuration files](#).

Now that you have a custom client configuration file, you can edit it to add your client details:

```
{
  "ClientId": "myid.mywebsite",
  "ClientName": "My Website",
  "AccessTokenLifetime": "3600",
  "AllowedGrantTypes": [
    "authorization_code"
  ],
}
```

```
"RequireClientSecret": false,
"RequirePkce": true,
"AllowAccessTokensViaBrowser": true,
"RequireConsent": false,
"AllowedScopes": [
  "myid.rest.basic"
],
"RedirectUris": [
  "<website URL>"
],
"AllowedCorsOrigins": [
  "<website origin>"
]
}
```

Where:

- `<website URL>` – the location of the website to which the authentication server will return the authorization code. In this single-page example, this is the same URL as you use to request the authorization code.

If you have multiple web pages on your website to which you want the authentication server to return an authorization code, you can add them to this array, separated by commas; for example:

```
"RedirectUris": [
  "http://127.0.0.1:5500/",
  "http://127.0.0.1:5500/page1.html",
  "http://127.0.0.1:5500/page2.html"
],
```

For this worked example, however, you keep reusing the same `index.html` file, so if you are using the Live Server extension for Visual Studio Code as your local web server, you can use:

```
"RedirectUris": [
  "http://127.0.0.1:5500/"
],
```

- `<website origin>` – used for Cross-Origin Resource Sharing (CORS). If the web page that calls the authentication service is on a different web server from the authentication service, you must add the origin to this list.

Note: Make sure you use an origin, and not an URL, when configuring CORS. For example: `https://myserver/` is an URL, while `https://myserver` is an origin – there is no trailing slash on an origin.

For this worked example, if you are using the Live Server extension for Visual Studio Code as your local web server, you can use:

```
"AllowedCorsOrigins": [
  "http://127.0.0.1:5500"
```

]

For example, assuming you are using the Live Server extension for Visual Studio Code, the `<website URL>` and `<website origin>` are as follows:

```
{
  "ClientId": "myid.mywebsite",
  "ClientName": "My Website",
  "AccessTokenLifetime": "3600",
  "AllowedGrantTypes": [
    "authorization_code"
  ],
  "RequireClientSecret": false,
  "RequirePkce": true,
  "AllowAccessTokensViaBrowser": true,
  "RequireConsent": false,
  "AllowedScopes": [
    "myid.rest.basic"
  ],
  "RedirectUris": [
    "http://127.0.0.1:5500/"
  ],
  "AllowedCorsOrigins": [
    "http://127.0.0.1:5500"
  ]
}
```

11.2.2 Configuring the MyID Core API for CORS

In the configuration for the web.oauth2 web service, you added the origin of your sample website to the `AllowedCorsOrigins` setting. You must also configure the rest.core service to allow the same origin.

In a text editor, open the `appsettings.Production.json` file for the rest.core service.

By default, this is:

`C:\Program Files\Intercede\MyID\rest.core\appsettings.Production.json`

This file is the override configuration file for the `appsettings.json` file for the web service. If this file does not already exist, you must create it in the same folder as the `appsettings.json` file, and include the following:

```
{
  "MyID": {
    "Cors": {
      "AllowedOrigins": [
        "<website origin>"
      ]
    }
  }
}
```

where:

- `<website origin>` is the server from which you are going to call the API. You can add multiple origins if necessary.

Note: Make sure you use an origin, and not an URL, when configuring CORS. For example: `https://myserver/` is an URL, while `https://myserver` is an origin – there is no trailing slash on an origin.

If the `appsettings.Production.json` file already exists, add the above to the `Cors:AllowedOrigins` section of the file.

For example:

```
{
  "MyID": {
    "Cors": {
      "AllowedOrigins": [
        "http://127.0.0.1:5500"
      ]
    }
  }
}
```

11.2.3 Refreshing the web server settings

Once you have made your changes to the web.oauth2 and rest.core server settings, you must refresh the application pools to ensure that all the systems are using the latest settings.

To refresh the server settings:

1. Recycle the web service app pools:
 - a. On the MyID web server, in Internet Information Services (IIS) Manager, select **Application Pools**.
 - b. Right-click the **myid.web.oauth2.pool** application pool, then from the pop-up menu click **Recycle**.
 - c. Right-click the **myid.rest.core.pool** application pool, then from the pop-up menu click **Recycle**.

This ensures that the web services have picked up the changes to the configuration file.

2. Check that the web.oauth2 and rest.core servers are still operational by logging on to the MyID Operator Client.

Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

```
HTTP Error 500.30 - ANCM In-Process Start Failure
```

11.3 Obtaining an authorization code

The first thing you must do when calling the API is to obtain an authorization code. To do this, you submit information to the `/connect/authorize` endpoint of the `web.oauth2` web service, which then allows you to use whatever authentication methods you have configured to identify yourself to the MyID authentication server; the authentication server then returns the authorization code.

See section [4.2.2, Requesting an authorization code](#) for an overview of this process.

The following example uses a visible HTML form so that you can see what is going on.

Create a new HTML document called `index.html`.

Add the following:

```
<html>
<head>
  <title>Single-page PKCE client authentication through Javascript</title>
</head>
<body>
  <div id="intro">
    <p>This single-page example uses client authentication to obtain an authorization
code.</p>
  </div>
</body>
</html>
```

After the `intro` div, add a div for the form:

```
<div id="login">
  <p>Click <b>Login</b> to authenticate to the MyID server using
your client credentials.</p>
  <form
    method="post"
    enctype="application/x-www-form-urlencoded"
    action="https://myserver/web.oauth2/connect/authorize">
    <p>Client id: <input type="text" name="client_id" value="myid.mywebsite"></p>
    <p>Scope: <input type="text" name="scope" value="myid.rest.basic"></p>
    <p>Redirect: <input type="text" name="redirect_
uri" value="http://127.0.0.1:5500/"></p>
    <p>Response Type: <input type="text" name="response_type" value="code"></p>
    <p>Code Challenge: <input type="text" name="code_challenge" value="lzKaVv4bWu06z_
m0yFynJj6zttnU5gYpXah8tLYKzGg"></p>
    <p>Code Challenge Method: <input type="text" name="code_challenge_
method" value="S256"></p>
    <input type="submit" value="Login">
  </form>
</div>
```

The form contains the following elements:

- `action` – the URL of the `/connect/authorize` endpoint on your MyID server; for example:

```
https://myserver/web.oauth2/connect/authorize
```

Important: Change `myserver` to the address of your own MyID server.

- `client_id` – the client ID you set up in the `web.oauth2` custom client configuration file. In this example, this is:

```
myid.mywebsite
```

See section [11.2.1, Adding the API client](#) for details.

- `scope` – the scope that you are requesting that the user gets. This must match the scope set up in the `web.oauth2` custom client configuration file.

In this example, this is:

```
myid.rest.basic
```

- `redirect_uri` – the page to which you want the authentication server to return the authorization code. This is a single-page website, so the URL is the same as the current page; in this case, using the Live Server extension:

```
http://127.0.0.1:5500/
```

- `response_type` – the type of authorization response returned. In this worked example, we use authorization codes (instead of, for example, tokens), so you must set this to:

```
code
```

- `code_challenge` – the PKCE code challenge. In this case, we are using a static code challenge for simplicity; in a later example, we will generate a random code verifier and derive a code challenge from it.

- `code_challenge_method` – set this to `S256`, indicating that we are using a SHA-256 hash.

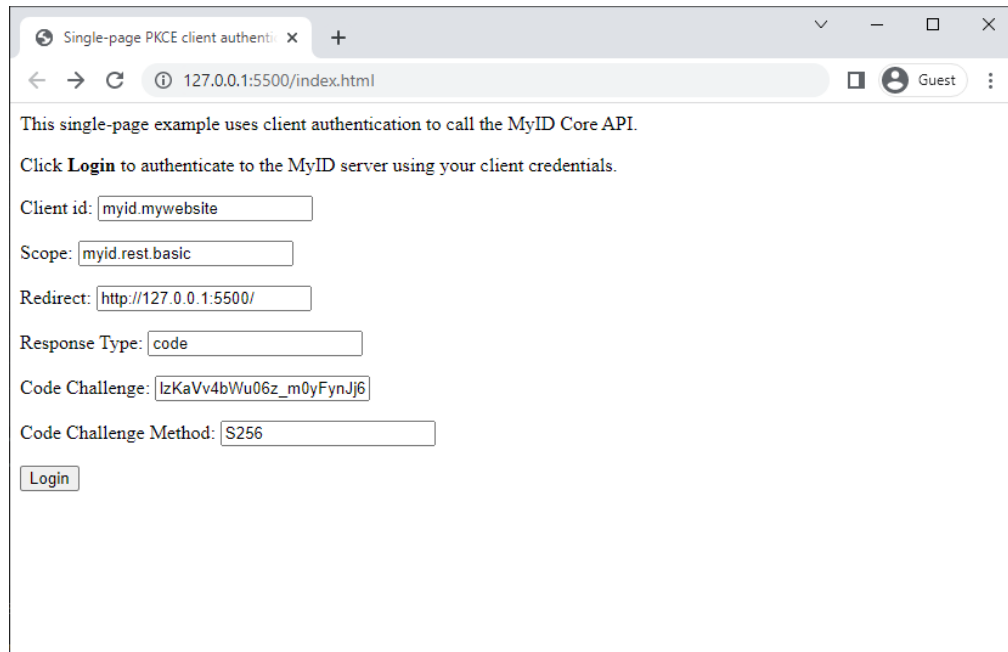
After the `login` div, add a script that checks for a returned authorization code, and replaces the form with a display of the code if it is available:

```
<script>
  // Get the authorization code from the URL parameters.
  const queryString = window.location.search;
  const urlParams = new URLSearchParams(queryString);
  const code = urlParams.get('code');

  // If the code is available, display it instead of the login form
  if (code) {
    document.getElementById("login").innerHTML="<p>The authorization code is: " +
code;
  }
</script>
```

You can now open this HTML page on your local web server.

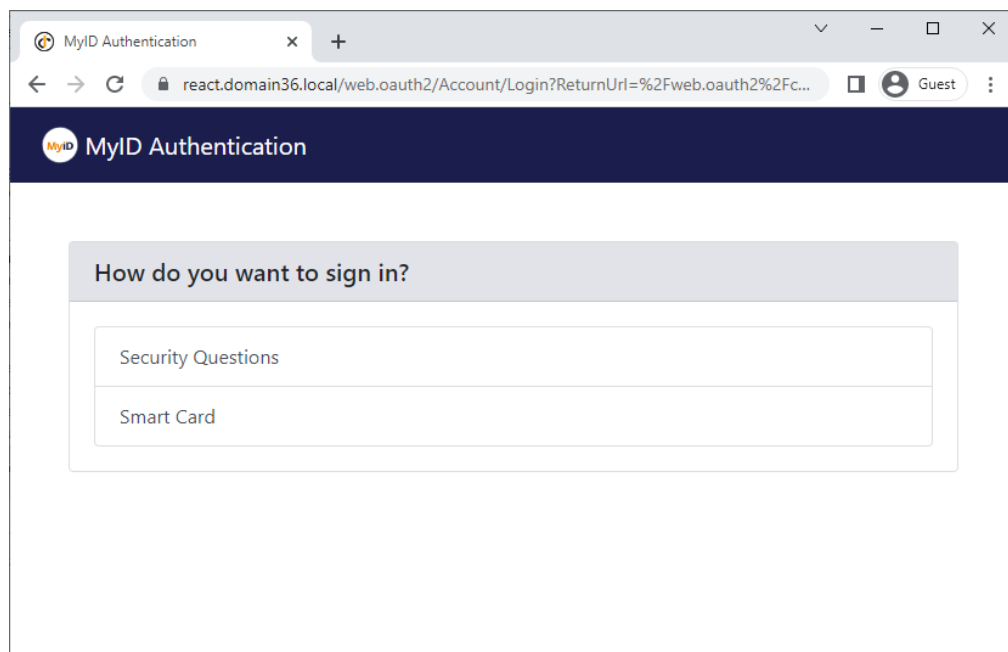
Using Visual Studio Code and the Live Server extension, right-click on the HTML file in the Explorer pane, then from the pop-up menu select **Open with Live Server**.



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:5500/index.html". The page content includes the following text and form fields:

- Text: "This single-page example uses client authentication to call the MyID Core API."
- Text: "Click **Login** to authenticate to the MyID server using your client credentials."
- Form field: "Client id:" with the value "myid.mywebsite".
- Form field: "Scope:" with the value "myid.rest.basic".
- Form field: "Redirect:" with the value "http://127.0.0.1:5500/".
- Form field: "Response Type:" with the value "code".
- Form field: "Code Challenge:" with the value "IzKaVv4bWu06z_m0yFynJj6".
- Form field: "Code Challenge Method:" with the value "S256".
- Form field: "Login" button.

Click **Login**, and you are taken to the MyID Authentication web page.

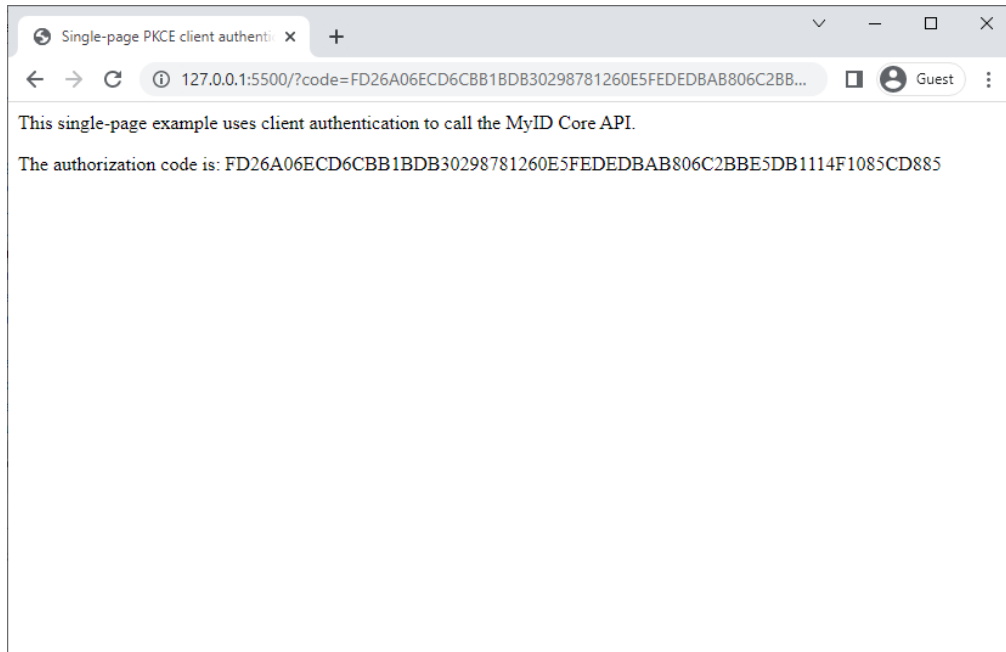


The screenshot shows the MyID Authentication web page. The header is dark blue with the MyID logo and the text "MyID Authentication". The main content area has a light gray background and contains the following text and form fields:

- Text: "How do you want to sign in?"
- Form field: "Security Questions".
- Form field: "Smart Card".

You can sign in with any configured authentication methods; in the above example, the MyID server is set up for security questions or smart cards, but you may have configured your system for external identity providers, authentication codes, FIDO, Windows Hello, integrated Windows login; any method you have configured for MyID is available.

Once you have authenticated, you are returned to the page specified in the `redirect_uri` parameter from the form. The web page obtains the authorization code from the URL and displays it on screen:



Obtaining an authorization code is the first step to calling the API. But before we progress further, we should make the request form a bit easier to work with.

11.4 Creating a customizable form

The previous example (see section [11.3, Obtaining an authorization code](#)) used a simple visible HTML form to request an authorization code. For the remaining examples, we are going to need to be able to change parts of the request form dynamically, so before progressing any further, we will use a script to create the form rather than using simple HTML.

This example is functionally identical to the previous example.

Remove the `index.html` from the previous example, and create a new HTML document called `index.html`.

```
<html>
<head>
  <title>Single-page PKCE client authentication through Javascript</title>
</head>
<body>
  <div id="intro">
    <p>This single-page example uses client authentication to obtain an authorization
code.</p>
  </div>
</body>
</html>
```

After the `intro` div, add a div for the form:

```
<div id="login">
  <p>Click <b>Login</b> to authenticate to the MyID server using
your client credentials.</p>
</div>
```

For now, the `login` div contains placeholder text. We will add the form using a script later in this example.

After the `login` div, add a `<script>` block and include the constants that we will use to build the form.

```
<script>
  // Address of the MyID server
  const server = "https://myserver";
  // Name of the client ID you have set up in the web.oauth2 appsettings file
  const client_id = "myid.mywebsite";
  // Scope configured for the client
  const scope = "myid.rest.basic";
  // The URL for this page - this must match a value for RedirectUri
  // in the web.oauth2 appsettings file.
  const redirect_uri = "http://127.0.0.1:5500/";
  // MyID oauth2 authorization URL
  const authorize_url = "/web.oauth2/connect/authorize";
  // Code challenge
  const code_challenge = "lzKaVv4bWu06z_m0yFynJj6zttnU5gYpXah8tLYKzGg";
</script>
```

These are the same values you included in the `<form>` in the previous example.

Important: Remember to change `myserver` to the address of your own MyID server.

Within the `<script>` block, add a script that checks for a returned authorization code, displays the code if it is available, and if not, creates the form:

```
// Get the authorization code from the URL parameters.
const queryString = window.location.search;
const urlParams = new URLSearchParams(queryString);
const code = urlParams.get('code');

// If the code is available, display it instead of the login form
if (code) {
  document.getElementById("login").innerHTML="<p>The authorization code is: " + code +
"</p>";
}
else {
  // Create the form
  createForm();
}
```

Now you need to add the function that creates the form within the `<script>` block. This is functionally identical to the form that you created using HTML in the first example, but instead of displaying all the parameters, it hides everything except the **Login** button.

```
function createForm() {
    // Create the form
    var form = document.createElement("form");
    form.setAttribute("method", "post");
    form.setAttribute("enctype", "application/x-www-form-urlencoded");
    form.setAttribute("action", server + authorize_url);

    // Create the hidden fields
    var client_id_element = createFormField("hidden", "client_id", client_id);
    var scope_element = createFormField("hidden", "scope", scope);
    var redirect_uri_element = createFormField("hidden", "redirect_uri", redirect_uri);
    var response_type_element = createFormField("hidden", "response_type", "code");
    var code_challenge_element = createFormField("hidden", "code_challenge", code_
challenge);
    var code_challenge_method_element = createFormField("hidden", "code_challenge_method",
"S256");

    // Create a Login button
    var submit_button = document.createElement("input");
    submit_button.setAttribute("type", "submit");
    submit_button.setAttribute("value", "Login");

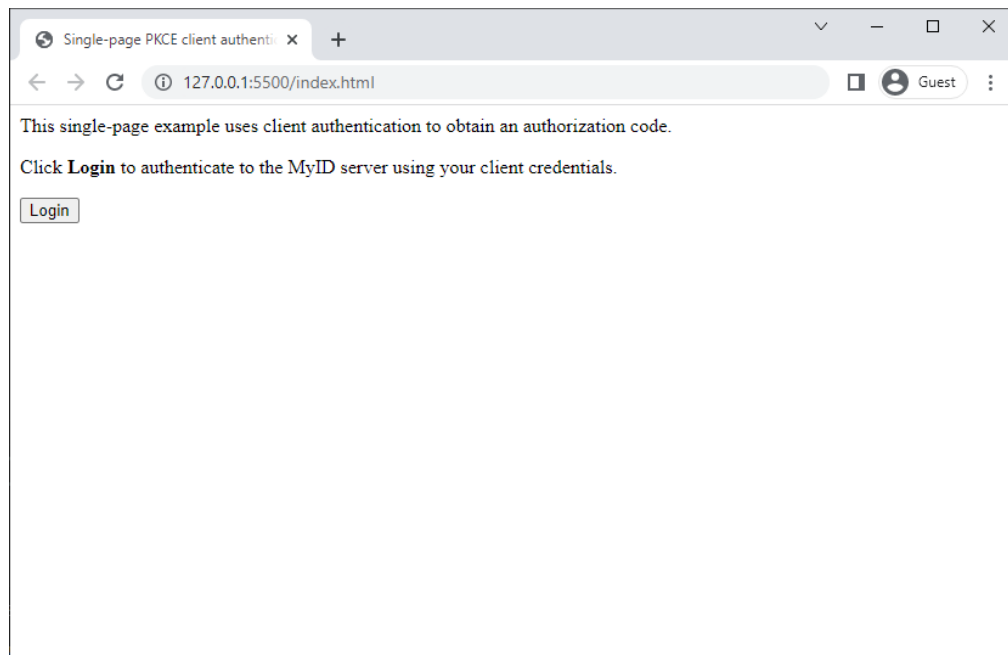
    // Append the fields to the form
    form.appendChild(client_id_element);
    form.appendChild(scope_element);
    form.appendChild(redirect_uri_element);
    form.appendChild(response_type_element);
    form.appendChild(code_challenge_element);
    form.appendChild(code_challenge_method_element);

    // Append the Login button to the form
    form.appendChild(submit_button);

    // Add the form to the login div
    document.getElementById("login").appendChild(form);
}

function createFormField(type, name, value) {
    var formFieldElement = document.createElement("input");
    formFieldElement.setAttribute("type", type);
    formFieldElement.setAttribute("name", name);
    formFieldElement.setAttribute("value", value);
    return formFieldElement;
}
```

You can now open this HTML page on your local web server.



Try it out. It works in the same way as the previous example: you click **Login**, then it takes you to the MyID Authentication web page. Once you've authenticated, the page displays the authorization code.

11.5 Obtaining an access token

Obtaining an authorization code is just the first stage. Now, you must use it to obtain an access token; only then can you call the API.

In this example, we are going to extend the web page you created in the previous example; see section 11.4, *Creating a customizable form*.

Edit the `index.html` that you created in the previous example, and change the description in the `intro` div, as this example now obtains an access token, not just an authorization code.

```
<div id="intro">
  <p>This single-page example uses client authentication to obtain an
  access token.</p>
</div>
```

Inside the `<script>` block, extend the list of constants.

```
// MyID oauth2 token URL
const token_url = "/web.oauth2/connect/token";
// Code verifier
const code_verifier = "TiGVEDHIRkdTpif4zLw8v6tcdG2VJXvP4r0fuLhsXIj";
```

Previously you needed the authorization URL and the PKCE code challenge; to obtain the access token, you now also need the following:

- `token_url` – this is the URL of the token endpoint of the MyID authentication server.
- `code_verifier` – this is the PKCE code verifier, the counterpart of the code challenge. You provide the code challenge when you request the authorization code, and when you use the authorization code to request the access token, you pass the corresponding code verifier; the authentication server makes sure that they match before returning the access token.

Immediately after the constants, create a new HTTP request object that you will use to request the access token:

```
// Set up the HTTP requests
// The request object is used to obtain an access token
var request = new XMLHttpRequest();
```

In the previous example, if the authorization code was present, the web page displayed it on screen. Instead, you want to use the code to request an access token, so edit the `if (code)` section of the script as follows:

```
// If the code is available, use it to obtain an access token
if (code) {
  getAccessToken();
}
else {
  // Create the form
  createForm();
}
```

Now you need to create the `getAccessToken()` function at the end of the `<script>` block:

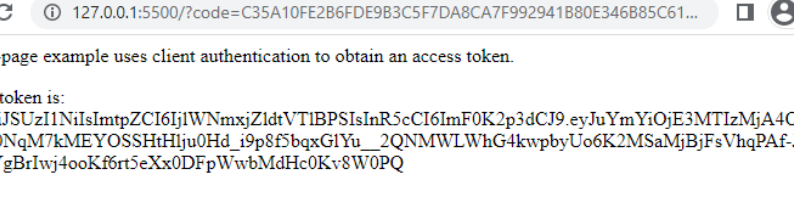
```
function getAccessToken() {  
    // Use the authorization code to obtain an access code  
    // This retrieves the code verifier from the browser's session storage  
    request.open("POST", server + token_url, true);  
    request.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');  
    request.send('client_id=' + client_id +  
        '&code_verifier=' + code_verifier +  
        '&grant_type=authorization_code&redirect_uri=' + redirect_uri +  
        '&code=' + code);  
    document.getElementById("login").innerHTML="Working...";  
}
```

This function uses the HTTP request object that you created above to POST a request to the token endpoint. It passes in the following:

- `client_id` – the client ID you set up in the web.oauth2 custom client configuration file.
In this example, this is:
`myid.mywebsite`
See section [11.2.1, Adding the API client](#) for details.
- `code_verifier` – the PKCE code verifier, the counterpart of the code challenge you used to create the request for the authorization code.
- `grant_type` – specifies the type of grant; in this case, to obtain an access token from an authorization code, use:
`authorization_code`
- `redirect_uri` – the page to which you want the authentication server to return the access token.
- `code` – the authorization code.

The function also replaces the form with a "Working..." placeholder while it is waiting for a response from the server.

```
// When the access token is returned, obtain it from the response.
request.onload = function() {
    let response = JSON.parse(request.responseText);
    let access_token = response.access_token;
    document.getElementById("login").innerHTML = "<p>The access token is: " + access_token +
"</p>";
}
```



Single-page PKCE client authentication x +

127.0.0.1:5500/?code=C35A10FE2B6FDE9B3C5F7DA8CA7F992941B80E346B85C61...

Guest

This single-page example uses client authentication to obtain an access token.

The access token is:

eyJhbGciOiJSUzI1NiIsImtpZCI6IjIWNmxiZldtVTI0PSIsInR5cCI6ImF0K2p3dCJ9.eyJ0eXciOiJ0e3MTIzMTJmV4cC5jcBI_T960NqM7kMEYOSShtHlju0Hd_i9p8f5bqxG1Yu__2QNMWLWhG4kwpbyUo6K2MSaMjBjFsVhqPAf-J8ihFjyRfva-c4Sf7FIJnYgBrlwj4ooKf6rt5eXx0DFpWwbMdHc0Kv8W0PQ

11.6 Generating a random code verifier and challenge

Before we go any further, we should take a look at the PKCE code verifier and code challenge.

Up to this point, we have been using a static code verifier and corresponding code challenge. This has let us get up and running, but the point of the PKCE system is to allow the authentication server to verify that the system that passed the request for the authorization code is the same system that is subsequently requesting the access token. To implement this, we must use a fresh, random code verifier each time.

For more information about PKCE code verifiers and code challenges, see section [4.2.1, Generating a PKCE code verifier and code challenge](#).

Edit the `index.html` file you created in the previous example (see section [11.5, Obtaining an access token](#)).

In the list of constants, *delete* the following lines:

```
// Code challenge
const code_challenge = "1zKaVv4bWu06z_m0yFynJj6ztttU5gYpXah8tLYKzGg";

// Code verifier
const code_verifier = "TiGVEDHIRkdTpif4zLw8v6tcdG2VJXvP4r0fuLhsXIj";
```

In the `createForm()` function, edit the line that creates the `code_challenge_element` of the form; change:

```
var code_challenge_element = createFormField("hidden", "code_challenge", code_challenge);
```

to:

```
var code_challenge_element = createChallengeField();
```

At the end of the `<script>` block, add the functions that you are going to use to create the code verifier and code challenge.

First, create a function to generate a random string:

```
// Create a function to generate a random code verifier string
function generateRandomString(length) {
  var text = "";
  var possible = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
  for (var i = 0; i < length; i++) {
    text += possible.charAt(Math.floor(Math.random() * possible.length));
  }
  return text;
}
```

This function creates a string of the specified length using the characters A-Z, a-z, and 0-9. You are going to use this to create the code verifier.

Next, create a function that generates a SHA-256 hash of the random string.

```
// Create an asynchronous function to generate a code challenge from the code verifier
async function generateCodeChallenge(code_verifier) {
  var digest = await crypto.subtle.digest("SHA-256",
    new TextEncoder().encode(code_verifier));
  return btoa(String.fromCharCode(...new Uint8Array(digest)))
    .replace(/=/g, '').replace(/\+/g, '-').replace(/\//g, '_')
}
```

This function takes the random code verifier, then creates a hash that you can use as the code challenge.

Next, create a function that uses the two functions above to create a code verifier and code challenge, which is then added to the code challenge on the request form.

```
function createChallengeField() {
  // Create the code challenge element for the form
  var code_challenge_element = document.createElement("input");
  code_challenge_element.setAttribute("type", "hidden");
  code_challenge_element.setAttribute("name", "code_challenge");

  // Create a random string to use as the code verifier
  var code_verifier = generateRandomString(128);
  // Store the code verifier to use later
  window.sessionStorage.setItem("code_verifier", code_verifier);

  // Generate the code challenge from the asynchronous function
  (async () => {
    const code_challenge = await generateCodeChallenge(code_verifier);
    code_challenge_element.setAttribute("value", code_challenge);
  })();

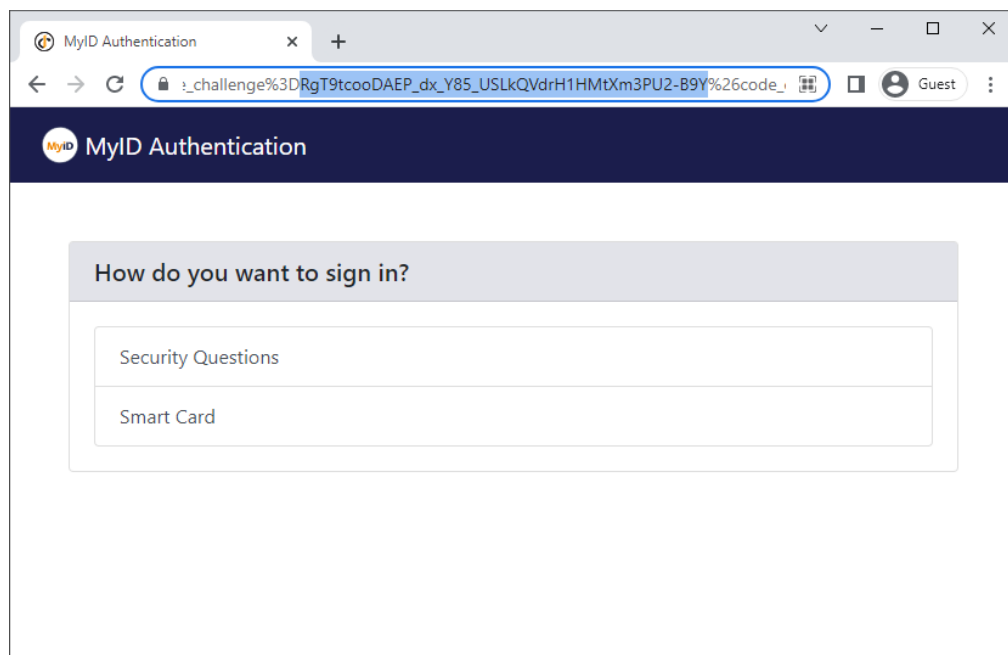
  // Return the code challenge form element
  return code_challenge_element;
}
```

Notice how the function stores the code verifier in the window session storage so we can use it later when requesting the access token.

Finally, edit the existing `getAccessToken()` function to retrieve the code verifier from the window session storage, and pass it to the token endpoint:

```
function getAccessToken() {  
    // Use the authorization code to obtain an access code  
    // This retrieves the code verifier from the browser's session storage  
    request.open("POST", server + token_url, true);  
    request.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');  
    request.send('client_id=' + client_id +  
        // Obtain the code_verifier from the session storage  
        '&code_verifier=' + window.sessionStorage.getItem("code_verifier") +  
        '&grant_type=authorization_code&redirect_uri=' + redirect_uri +  
        '&code=' + code);  
    document.getElementById("login").innerHTML="Working...";  
}
```

Try it out. It should work in exactly the same way as the previous example, with the only difference being you are securely generating a random code verifier and deriving a corresponding code challenge each time. You can verify this by checking the URL passed to the web.oidc service, which contains the code challenge:



Now we can move on to the main event – calling the API.

11.7 Calling the API

The previous example generated a random code verifier and code challenge, which allowed us to request an authorization code, and from that to request an access token.

Now that we have our access token, we can use it to call the API. For our first step, we are going to get a list of the people in the MyID database.

Important: The access you have to the MyID system is determined by the user account you use to authenticate – that user account must have permissions to access the appropriate endpoint in the API, and must have scope to retrieve information about other people in the MyID system. See section 2.2, [Accessing the API features](#) for more information.

Edit the `index.html` file you created in the previous example (see section 11.6, [Generating a random code verifier and challenge](#)).

Edit the introduction:

```
<div id="intro">
  <p>This single-page example uses client authentication to obtain an
    access token and call the API.</p>
</div>
```

In the section of the script where you created the HTTP request object that you used to obtain the access token, add a new HTTP request object that you can use to call the API.

```
// Set up the HTTP requests
// The request object is used to obtain an access token
var request = new XMLHttpRequest();
// The call_api object is used to call the API
var call_api = new XMLHttpRequest();
```

In the previous example, when the request object returned its results, the `request.onload` event simply displayed the access token on screen. You can now extend that event function to make use of the access token:

```
// When the access token is returned, obtain it from the response.
request.onload = function() {
  let response = JSON.parse(request.responseText);
  let access_token = response.access_token;
  window.sessionStorage.setItem("access_token", access_token);
  getPeople();
}
```

Instead of displaying the access token, the function now stores it in the window session storage, then calls `getPeople()`.

At the end of the `<script>` block, add the `getPeople` function:

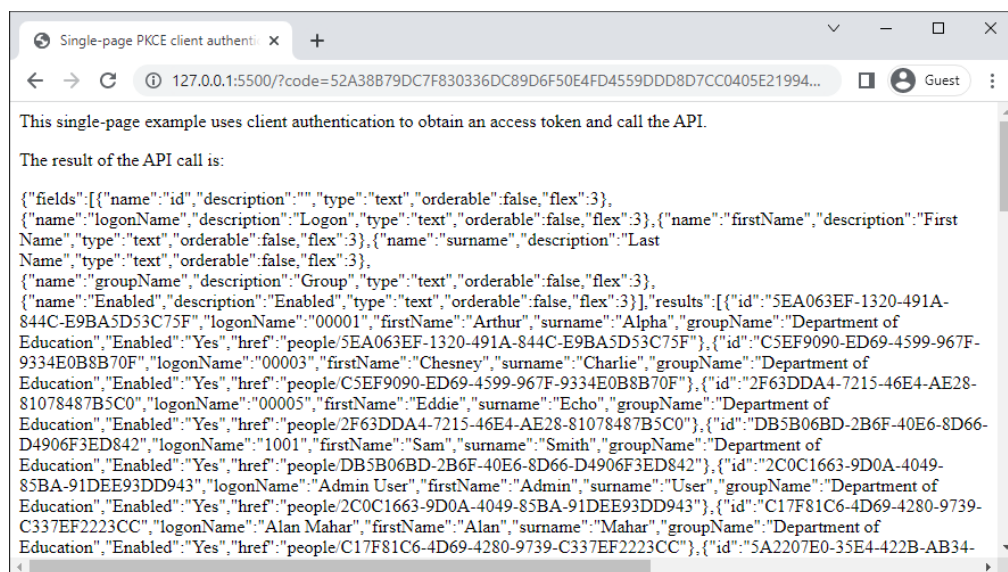
```
function getPeople() {
  call_api.open("GET", server + "/rest.core/api/People?q=", true);
  call_api.setRequestHeader('Authorization', "Bearer " + window.sessionStorage.getItem("access_token"));
  call_api.send();
}
```

This function uses the `call_api` HTTP request object that you created earlier to call the API, using the access token from the window session storage.

Now you need to add a handler for the response from the `call_api` HTTP request object.

```
call_api.onload = function() {
  document.getElementById("login").innerHTML = "<p>The result of the API call is:</p><p>"
+
  call_api.responseText + "</p>";
}
```

We are just going to display the raw response for now. Try it out.



The response from the `/api/People` endpoint is in JSON format. This does not look great on screen, so in the next example we are going to attempt to process it into a human-readable list.

11.8 Displaying a list of people

The previous example called the API to return a list of people, but the results were in raw JSON format, so in this next example, we are going to process the JSON to display a list of the names of people in the MyID system.

Edit the `index.html` file you created in the previous example (see section [11.7, Calling the API](#)).

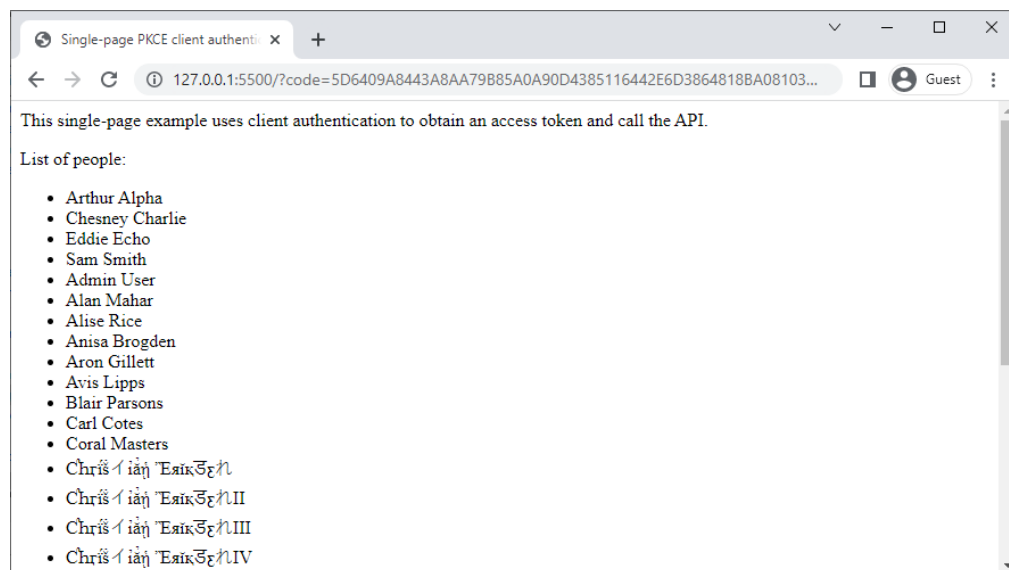
Edit the `call_api.onload` event, and instead of displaying the raw data, process the JSON and return it as an HTML list.

```
call_api.onload = function() {
  let response = JSON.parse(call_api.responseText);
  message = "<p>List of people:</p><ul>";
  for (key in response.results) {
    message += "<li>" +
      response.results[key].firstName + " " +
      response.results[key].surname + "</li>";
  }
  message += "</ul>";
  document.getElementById("login").innerHTML = message;
}
```

This function now parses the JSON returned from the API. The JSON contains a great deal of information, which you can review at your leisure; you can obtain the output from the previous example and run it through a JSON formatter to allow you to study its structure. There is also more information about the responses from each endpoint in the Swagger API documentation, see section 2.1, [Accessing the API documentation](#) for more information on how to access that documentation.

For now, we are interested only in the `results` node, and within that, only the person's name. The function iterates over each result in the JSON and finds the first name and surname of the person, and constructs a list from this information, which it then writes to the screen.

Try it out.



Next, we will get more information about each person by using their unique ID.

11.9 Displaying the details for a person

In the previous example, when we obtained the list of people, we took only the person's name from the results returned from the API. There is one more bit of crucial information that the API returned, and we are going to need this for the next example – the ID.

The MyID Core API uses unique IDs throughout; they are how you refer to a person, a group, a credential profile, a request, and so on. Each ID is a GUID; for example:

5EA063EF-1320-491A-844C-E9BA5D53C75F

The ID for each person is returned along with the rest of the information about the person in the array of results that we have obtained; for example:

```
{
  "results": [
    {
      "id": "5EA063EF-1320-491A-844C-E9BA5D53C75F",
      "logonName": "00001",
      "firstName": "Arthur",
      "surname": "Alpha",
      "groupName": "Department of Education",
      "Enabled": "Yes",
      "href": "people/5EA063EF-1320-491A-844C-E9BA5D53C75F"
    }
  ]
}
```

Edit the `index.html` file you created in the previous example (see section [11.8, Displaying a list of people](#)).

In the section where you created the HTTP request objects that you used to obtain the access token and to call the API, add a new HTTP request object that you can use to call the API..

```
// Set up the HTTP requests
// The request object is used to obtain an access token
var request = new XMLHttpRequest();
// The call_api object is used to call the API
var call_api = new XMLHttpRequest();
// The display_person object is used to get info about a person
var display_person = new XMLHttpRequest();
```

Extend the handler for the `call_api.onload` event:

```
call_api.onload = function() {
  let response = JSON.parse(call_api.responseText);
  message = "<p>List of people:</p><ul>";
  for (key in response.results) {
    message += "<li style='cursor:pointer;' onclick='displayPerson(\"" +
      response.results[key].id + "\" )'>" +
      response.results[key].firstName + " " +
      response.results[key].surname + "</li>";
  }
  message += "</ul>";
  message += "<p style='cursor:pointer;' onclick='logout()'><b>Logout</b></p>";
  document.getElementById("login").innerHTML = message;
}
```

In addition to the first and last name, this function adds an `onclick` event to each list item that calls the `displayPerson` function with the ID of the relevant person.

(This function also adds a **Logout** button which we will cover later in this example.)

At the end of the `<script>` block, add the `displayPerson` function:

```
function displayPerson(personID) {
  display_person.open("GET", server + "/rest.core/api/People/" + personID, true);
  display_person.setRequestHeader('Authorization', "Bearer " +
  window.sessionStorage.getItem("access_token"));
  display_person.send();
}
```

This function takes the ID that is passed in as a parameter, then uses the `display_person` HTTP request object to call the API to obtain a full list of information about the person.

Now, add a handler for the response from the `display_person` HTTP request object.

```
display_person.onload = function() {
  let response = JSON.parse(display_person.responseText);
  message = "<p><b>Name:</b> " + response.name.fullName + "</p>";
  message += "<p><b>Group:</b> " + response.group.name + "</p>";
  message += "<p><b>Logon Name:</b> " + response.logonName + "</p>";
  message += "<p><b>Roles:</b></p><ul>";
  for (key in response.roles) {
    message += "<li>" + response.roles[key].name + "</li>"
  }
  message += "</ul><p style='cursor:pointer;' onclick='getPeople()'><b> &lt; Back</b></p>";
  document.getElementById("login").innerHTML = message;
}
```

This function parses the JSON response from the API call and pulls out the name, group, and logon name. This information was already available to us in the original request to obtain a list of people, but now we can dig a bit deeper, and pull out a list of all of the roles that the person is assigned.

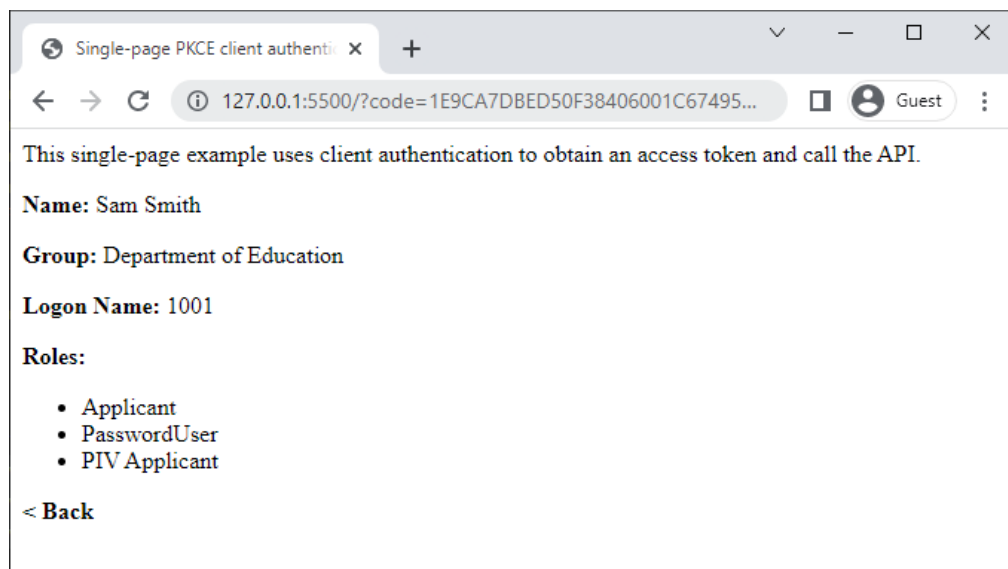
Because of the simplistic way we have coded this example, the back button in the browser will not work, so we can add our own **Back** link to take us back to the list of people; it simply calls the `getPeople` function again.

As one last bit of tidying up, we can add a function for the **Logout** button we added earlier.

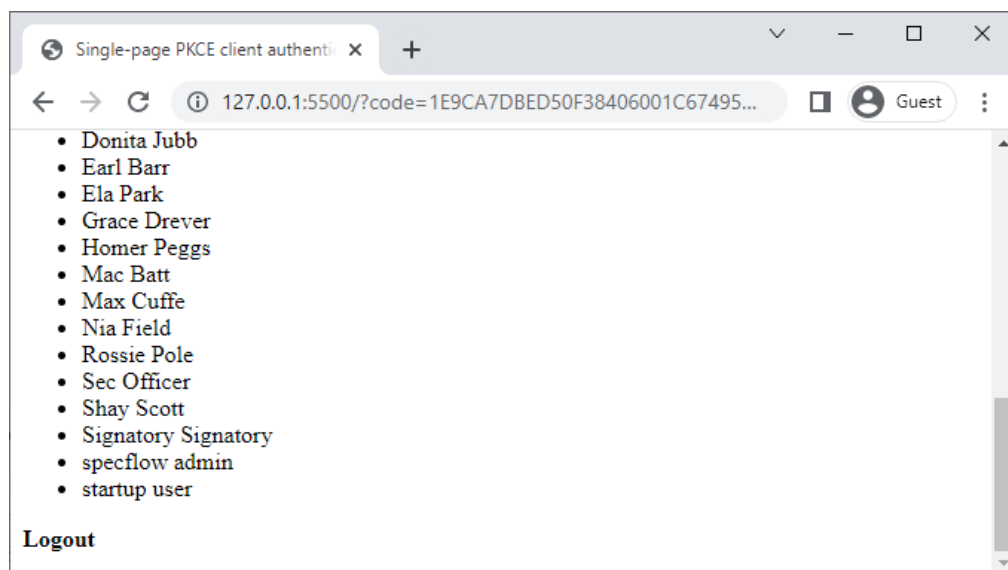
```
logout = function() {  
  window.sessionStorage.setItem("access_token", "");  
  window.sessionStorage.setItem("code_verifier", "");  
  window.location.href="/";  
}
```

This function removes the access token and the code verifier from the window session storage, and returns us to the login screen. This lets you run through the whole process of authentication and selecting a person more easily.

Try it out. You can now click on a person's name, and display further details about that person, including their roles.



You can also click **Back** from the person's details, then click **Logout** to return to the login screen.



11.9.1 Exploring the links

If you look at the JSON returned from the request for the person's details, after all of the personal information is a `links` block. This provides links to every action you can carry out for that person.

For example:

```
{
  "op": "100111",
  "cat": "control",
  "desc": "Persons Available Credential Profiles",
  "verb": "GET",
  "href": "people/5EA063EF-1320-491A-844C-E9BA5D53C75F/credprofiles"
},
```

This lets you know how to obtain a list of the person's available credential profiles; you would make a call to:

```
GET /api/people/<id>/credprofiles
```

You could then use this information to request a device for the person, specifying a credential profile to which you know they have access.

This is your entry point to exploring further how you can use the API. Each entity (person, request, device, and so on) is linked to other entities, and you can use the MyID Core API to navigate this network to manage many aspects of your MyID system and the lifecycle of the credentials you need to issue.

Further exploration of the API is beyond the scope of this brief tutorial, and is left as an exercise for the reader.

11.10 Example code listing

The following is the full listing of the final example in this chapter.

To use this on your own system, you must edit the following:

```
// Address of the MyID server
const server = "https://myserver";
// The URL for this page - this must match a value for RedirectUri
// in the web.oidc appsettings file.
const redirect_uri = "http://127.0.0.1:5500/";
```

- Set the `server` option to the URL of your MyID server.
- Set the `redirect_uri` to the URL of this sample web page on your local web server; if you are using the Live Server extension for Visual Studio Code, this is:

`http://127.0.0.1:5500/`

```
<html>
<head>
  <title>Single-page PKCE client authentication through Javascript</title>
</head>
<body>
  <div id="intro">
    <p>This single-page example uses client authentication to obtain an
    access token and call the API.</p>
  </div>
  <div id="login">
    <p>Click <b>Login</b> to authenticate to the MyID server using
    your client credentials.</p>
  </div>
  <script>
    // Address of the MyID server
    const server = "https://myserver";
    // Name of the client ID you have set up in the web.oidc appsettings file
    const client_id = "myid.mywebsite";
    // Scope configured for the client
    const scope = "myid.rest.basic";
    // The URL for this page - this must match a value for RedirectUri
    // in the web.oidc appsettings file.
    const redirect_uri = "http://127.0.0.1:5500/";
    // MyID oidc authorization URL
    const authorize_url = "/web.oidc/connect/authorize";
    // MyID oidc token URL
    const token_url = "/web.oidc/connect/token";

    // Set up the HTTP requests
    // The request object is used to obtain an access token
    var request = new XMLHttpRequest();
    // The call_api object is used to call the API
    var call_api = new XMLHttpRequest();
    // The display_person object is used to get info about a person
    var display_person = new XMLHttpRequest();

    // Get the authorization code from the URL parameters.
    const queryString = window.location.search;
    const urlParams = new URLSearchParams(queryString);
    const code = urlParams.get('code');
```

```
// If the code is available, use it to obtain an access token
if (code) {
    getAccessToken();
}
else {
    // Create the form
    createForm();
}

function getAccessToken() {
    // Use the authorization code to obtain an access code
    // This retrieves the code verifier from the browser's session storage
    request.open("POST", server + token_url, true);
    request.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
    request.send('client_id=' + client_id +
        // Obtain the code_verifier from the session storage
        '&code_verifier=' + window.sessionStorage.getItem("code_verifier") +
        '&grant_type=authorization_code&redirect_uri=' + redirect_uri +
        '&code=' + code);
    document.getElementById("login").innerHTML="Working...";
}

// When the access token is returned, obtain it from the response.
request.onload = function() {
    let response = JSON.parse(request.responseText);
    let access_token = response.access_token;
    window.sessionStorage.setItem("access_token", access_token);
    getPeople();
}

function createForm() {
    // Create the form
    var form = document.createElement("form");
    form.setAttribute("method", "post");
    form.setAttribute("enctype", "application/x-www-form-urlencoded");
    form.setAttribute("action", server + authorize_url);

    // Create the hidden fields
    var client_id_element = createFormField("hidden", "client_id", client_id);
    var scope_element = createFormField("hidden", "scope", scope);
    var redirect_uri_element = createFormField("hidden", "redirect_uri", redirect_
uri);
    var response_type_element = createFormField("hidden", "response_type",
"code");
    var code_challenge_element = createChallengeField();
    var code_challenge_method_element = createFormField("hidden", "code_challenge_
method", "S256");

    // Create a Login button
    var submit_button = document.createElement("input");
    submit_button.setAttribute("type", "submit");
    submit_button.setAttribute("value", "Login");

    // Append the fields to the form
    form.appendChild(client_id_element);
    form.appendChild(scope_element);
    form.appendChild(redirect_uri_element);
    form.appendChild(response_type_element);
}
```

```
form.appendChild(code_challenge_element);
form.appendChild(code_challenge_method_element);

// Append the Login button to the form
form.appendChild(submit_button);

// Add the form to the login div
document.getElementById("login").appendChild(form);
}

function createFormField(type, name, value) {
  var formFieldElement = document.createElement("input");
  formFieldElement.setAttribute("type", type);
  formFieldElement.setAttribute("name", name);
  formFieldElement.setAttribute("value", value);
  return formFieldElement;
}

// Create a function to generate a random code verifier string
function generateRandomString(length) {
  var text = "";
  var possible =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
  for (var i = 0; i < length; i++) {
    text += possible.charAt(Math.floor(Math.random() * possible.length));
  }
  return text;
}

// Create an asynchronous function to generate a code challenge from the code
// verifier
async function generateCodeChallenge(code_verifier) {
  var digest = await crypto.subtle.digest("SHA-256",
    new TextEncoder().encode(code_verifier));
  return btoa(String.fromCharCode(...new Uint8Array(digest)))
    .replace(/=/g, '').replace(/\+/g, '-').replace(/\//g, '_')
}

function createChallengeField() {
  // Create the code challenge element for the form
  var code_challenge_element = document.createElement("input");
  code_challenge_element.setAttribute("type", "hidden");
  code_challenge_element.setAttribute("name", "code_challenge");

  // Create a random string to use as the code verifier
  var code_verifier = generateRandomString(128);
  // Store the code verifier to use later
  window.sessionStorage.setItem("code_verifier", code_verifier);

  // Generate the code challenge from the asynchronous function
  (async () => {
    const code_challenge = await generateCodeChallenge(code_verifier);
    code_challenge_element.setAttribute("value", code_challenge);
  })();

  // Return the code challenge form element
  return code_challenge_element;
}
```

```

function getPeople() {
    call_api.open("GET", server + "/rest.core/api/People?q=", true);
    call_api.setRequestHeader('Authorization', "Bearer " +
window.sessionStorage.getItem("access_token"));
    call_api.send();
}

call_api.onload = function() {
    let response = JSON.parse(call_api.responseText);
    message = "<p>List of people:</p><ul>";
    for (key in response.results) {
        message += "<li style=\"cursor:pointer;\" onclick=\"displayPerson(\"' +
            response.results[key].id + \"'\")\">\" +
            response.results[key].firstName + \" \" +
            response.results[key].surname + "</li>";
    }
    message += "</ul>";
    message += "<p style=\"cursor:pointer;\" onclick=\"logout
()\"><b>Logout</b></p>";
    document.getElementById("login").innerHTML = message;
}

function displayPerson(personID) {
    display_person.open("GET", server + "/rest.core/api/People/" + personID,
true);
    display_person.setRequestHeader('Authorization', "Bearer " +
window.sessionStorage.getItem("access_token"));
    display_person.send();
}

display_person.onload = function() {
    let response = JSON.parse(display_person.responseText);
    message = "<p><b>Name:</b> " + response.name.fullName + "</p>";
    message += "<p><b>Group:</b> " + response.group.name + "</p>";
    message += "<p><b>Logon Name:</b> " + response.logonName + "</p>";
    message += "<p><b>Roles:</b></p><ul>";
    for (key in response.roles) {
        message += "<li>\" + response.roles[key].name + "</li>\"
    }
    message += "</ul><p style=\"cursor:pointer;\" onclick=\"getPeople()\"><b>
    &lt; Back</b></p>";
    document.getElementById("login").innerHTML = message;
}

logout = function() {
    window.sessionStorage.setItem("access_token", "");
    window.sessionStorage.setItem("code_verifier", "");
    window.location.href="/";
}
</script>
</body>
</html>

```

12 Troubleshooting

If you experience problems when attempting to use the MyID Core API:

- Check that you have set up the server-to-server authentication correctly.
See section [3, Server-to-server authentication](#).
- For PKCE-based end-user authentication, check that you have set up PKCE correctly.
See section [4.2.1, Generating a PKCE code verifier and code challenge](#), and see the PKCE standard for details of requirements for the code verifier and code challenge:
tools.ietf.org/html/rfc7636
Make sure you are using Base64 URL encoding to create the code challenge, and not simply Base64 encoding.
- Review the documentation to ensure that you are using the API calls correctly.
See section [2.1, Accessing the API documentation](#).
- Check that you have configured access to the API for the features you want to use.
See section [2.2, Accessing the API features](#).
- If you cannot see the people, requests, devices and so on that you expect, check that the operator account used to access the API has the correct scope.
See section [2.2.1, Scope](#).
- If you have clients in the `appsettings.json` file *and* the `appsettings.Production.json` file, make sure the production file does not overwrite the entries in the base file. In these settings files, entries in arrays are determined by their index; therefore if you have four existing entries in the `appsettings.json` file, you must include four blank array entries `{}`, in the `appsettings.Production.json` file before you include your new client details. Alternatively, you can include the entire `Clients` array in the `appsettings.Production.json` file.
You are recommended instead to use individual custom client configuration files (using the `CustomClients` folder) instead of providing your client information in the `appsettings.json` or `appsettings.Production.json` files; see section [3.2, Configuring web.oidc for server-to-server authentication](#) and section [4.1, Configuring web.oidc for end-user based authentication](#).
- Application setting JSON files are sensitive to such issues as comma placement; if the format of the file is not correct, the web service cannot load the file and will not operate, which may result in an error similar to:

```
HTTP Error 500.30 - ANCM In-Process Start Failure
```

Note especially that copying code samples from a browser may include hard spaces, which cause the JSON file to be invalid.

To assist in tracking down the problem, you can use the Windows Event Viewer. Check the **Windows Logs > Application** section for errors; you may find an error from the .NET Runtime source that contains information similar to:

```
Exception Info: System.FormatException: Could not parse the JSON file.
--> System.Text.Json.JsonReaderException: '"' is invalid after a
value. Expected either ',', '}', or ']'. LineNumber: 13 |
BytePositionInLine: 6.
```

which could be caused by a missing comma at the end of a line.

An error similar to:

```
Exception Info: System.FormatException: Could not parse the JSON file.
--> System.Text.Json.JsonReaderException: '0xC2' is an invalid start
of a property name. Expected a '"'. LineNumber: 7 | BytePositionInLine:
0.
```

is caused by a hard (non-breaking) space copied from a web browser, which is not supported in JSON.

Note: Some JSON files used by MyID contain comment lines beginning with double slashes // – these comments are not supported by the JSON format, so the JSON files will fail validation if you attempt to use external JSON validation tools. However, these comments *are* supported in the JSON implementation provided by asp.net.core, and so are valid in the context of MyID.

- If you see an error message, look up the error code.
See the [Error Code Reference](#) guide for a list of errors codes, their causes, and potential solutions.
- Enable logging on the rest.core and web.oauth2 web services.
See the *MyID REST and authentication web services* section in the [Configuring Logging](#) guide.