Simple identity server documentation

Release 1.0

Simple identity server

March 21, 2016
# Contents

1 Terminologies 3

2 OpenId / OAUTH2.0 5
   2.1 Authorization code flow 5
   2.2 Implicit flow 7
   2.3 Resource owner credentials flow 7
   2.4 Client credentials flow 8
   2.5 Hybrid flow 9

3 Endpoints 11
   3.1 Authorization 11
   3.2 Token 12
   3.3 User information 12
   3.4 Discovery 13
   3.5 Json Web Keys 14
   3.6 Registration 14
   3.7 Introspection 14

4 Quick start 15
   4.1 In 5 steps 15
   4.2 Options 17

5 Docker 19
Contents:
Terminologies

The open-id architecture is organized around several components:

- **Client**: An application which wants to execute an operation on a protected API
- **Protected API**: A REST API service which protects operations and/or personal information
- **Resource Owner**: A user of the client. A protected API can contain personal information, a client which wants to access to the resource owner’s information needs his consent.
- **Simple Identity Server**: An application which grants a client access to a protected API

Two different kinds of token can be requested from Simple Identity Server:

- **Access token**: An access token has a lifetime and is valid for certain scopes. It’s used by the client to access to a certain protected operation or to fetch resource owner’s information from the “userinfo” endpoint.
- **Identity token**: Signed (Json Web Signature) and/or encrypted (Json Web Encryption) Javascript Web Token. It contains all the resource owner’s claims.

We identified two different kinds of scopes:

- **Resource owner scope**: Used to retrieve a set of resource owner’s claims. For example by using the scope `email` you can access to the claims “email” & “email_verified”.
- **Protected Api Scope**: Used to protect API operation(s). For example an API operation can be protected against malicious HTTP request(s) which doesn’t contain an access token valid for the scope “executeOperation”.

The following schema shows the interactions between components.

1. The client which wants to access to certain API operations requests an access token from Simple Identity Server.
2. An HTTP request with the access token passed in the Authorization header (Authorization: Bearer <access token>) is sent to the protected API.
3. When the request is received a validation procedure is triggered, it sends a request to Simple Identity Server to check the access token validity and retrieve the scopes and/or resource owner’s claims such as: name, email.
Chapter 1. Terminologies
A client uses the authorization grant to obtain an access token, which is a string representing an authorization granted to the client by the resource owner. The RFC-6749 is talking about four different grant-types, but in reality there’re five. An HybridFlow has been introduced by open-id. We are going to explain the different flows by trying to implement the same scenario:

A client wants to access to protected operations but only users with administrator role are allowed. Protected API operations are expecting to receive HTTP request with access-token valids for the scope “role”, it will be used later by the API to retrieve user’s claims from the userinfo endpoint.

**2.1 Authorization code flow**

This flow is perfect when your client is a web-application. It has the benefit of not exposing any tokens to the User-Agents. The difference between an access-token & the authorization code is their lifetimes. An authorization code has a very short-lifetime and can only be used one time, however an access token can be used until its expiration date. The flow is explained in the following schema:
Steps:

Assumptions [We assumed that the resource owner is already authenticated & the consent has already been confirmed by him.]

1. The user-agent is redirected to the authorization endpoint to get a code.
2. Parameters are checked by Simple Identity Server.
3. The authorization code is passed to the callback as a fragment and returned to the client.
4. When the authorization code has been received, the client is using it to retrieve an access token.
5. The access token is generated and returned to the client.
6. The client is using the access token to access to protected operations.
7. The protected API passed the access token to the userinfo request & gets the claims (role) back.
8. Claims are returned by Simple Identity Server. They are used to limit the access to protected operations based on the roles.
9. Claims are checked during the authorization phase.
2.2 Implicit flow

With the implicit flow you can choose between having an identity token or an access token & identity token. In our scenario we are only interesting about retrieving an access token. The flow is explained in the following schema:

Steps:

Assumptions [We assumed that the resource owner is already authenticated & the consent has already been confirmed by him.]

1. The user-agent is redirected to the authorization endpoint to get an identity token & token
2. Parameters are checked by Simple Identity Server.
3. The identity token & token are passed to the callback as fragments and returned to the client.
4. The client is using the access token to access to protected operations.
5. The protected API passed the access token to the userinfo request & gets the claims (role) back.
6. Claims are returned by Simple Identity Server.
7. Claims are checked during the authorization phase.

2.3 Resource owner credentials flow

Using this flow doesn’t require resource owner’s interaction. This approach can be chosen when the client is a desktop application (WPF) & if there’s a trust relationship between the client and Simple Identity Server. The flow is explained in the following schema:
Steps:

1. Post the resource owner & client’s credentials to the token endpoint.
2. An access token valid for the scopes: profile & roles is returned to the client.
3. Token is passed into the API request.
4. The API is using the token to retrieve resource owner’s claims from Simple Identity Server.
5. Claims are returned.

2.4 Client credentials flow

The client-credentials flow has been introduced by OAuth2.0. It’s used to retrieve an access token valid for resource scopes. The scenario explained above cannot be supported. In fact the access token cannot be valid for resource owner scopes such as: roles and profile. This flow is used when the client is an API which wants to access to protected API operations. the flow is explained in the following schema:
Steps:

1. Post client credentials to token endpoint.
2. An access token valid for getoperation is generated and returned to the client.
3. Token is passed into the API request.
4. The API is using the token to retrieve the token’s claims from Simple Identity Server
5. The valid scopes are returned to the client.

2.5 Hybrid flow

The hybrid flow is coming from the Open-Id concept, it’s a mix between implicit & authorization code flows. The client can ask in the same request an authorization code and any tokens. The combinations below can be passed into the “response_type” parameter:

- code id_token
- code token
- code id_token token
3.1 Authorization

The interaction with the Authorization endpoint is needed, when the client is trying to fetch a token via one of the three grant-types: authorization code, implicit or hybrid flow. Depending on certain conditions, the endpoint can returns to a callback URL used by the client: an access token, authorization code or identity token.

- **An access token is returned when**: The response_type is id_token token and the registered client supports the implicit grant type. The response_type is code token and the registered client supports the authorization_code & implicit grant types.

- **An identity token is returned when**: The response_type is id_token and the registered client supports the implicit grant type. The response_type is token id_token and the registered client supports the implicit & authorization_code grant types.

- **An authorization code is returned when**: The response_type is code and the registered client supports the authorization_code grant type.

Parameters are passed in the query-string:
### Token

**3.2 Token**

TO COMPLETE

### User information

**3.3 User information**

User information endpoint returns a set of claims about the authenticated user. The list is represented by a JSON object that contains a collection of name and value pairs for the Claims. An HTTP GET / POST request is sent by the client to the user information endpoint. An access-token obtained from Open-Id provider needs to be specified as a **BearerToken** otherwise an unauthorized error is returned.

There’re three different ways to pass an access token:

1. Normal way is by using the Authorization header.
2. Sending the access token in the HTTP request entity-body.
3. Passing it in the HTTP request URI.

Depending on the client configuration, claims can be returned either in a clear JSON object or a JWT (JWS / JWE). More information can be found in ClientRegistration documentation.
3.4 Discovery

Discovery endpoint is an indirection layer. It contains the required informations to interact with the OpenId provider (SimpleIdentityServer). According to Glenn Block author of the book “Designing Evolvable Web APIs with ASP.NET” the advantage of using this layer is: "By creating discovery documents at API entry points, we can enable clients to dynamically identify the location of certains resources without having to hardcode URIs into the client application".

The url of the discovery endpoint is `http://localhost/.well-known/openid-configuration`. When an HTTP GET request is executed against it, a JSON object is returned with the following properties. (For more information you can refer to the official OpenIdConnectDiscovery documentation)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>authorization_endpoint</td>
<td>URL of the Authorization endpoint</td>
<td>/authorization</td>
</tr>
<tr>
<td>check_session_iframe</td>
<td>URL of an OP iframe that supports cross-origin communications</td>
<td>NOT SUPPORTED</td>
</tr>
<tr>
<td>claims_parameter_supported</td>
<td>Boolean value specifying whether the OP supports use of the claims parameter</td>
<td>true</td>
</tr>
<tr>
<td>claims_supported</td>
<td>JSON array containing a list of Claims Names</td>
<td>[sub, name, family_name, given_name etc..]</td>
</tr>
<tr>
<td>end_session_endpoint</td>
<td>URL at the OP to which an RP can perform a redirect to request that the End-User be logged out</td>
<td>NOT SUPPORTED</td>
</tr>
<tr>
<td>grant_types_supported</td>
<td>JSON array containing a list of grant-types</td>
<td>[authorization_code, password, refresh_token, implicit]</td>
</tr>
<tr>
<td>id_tokenSigningAlgValues_supported</td>
<td>JSON supported containing a list of JWS signing algorithms</td>
<td>[RS256]</td>
</tr>
<tr>
<td>issuer</td>
<td>Issuer identifier</td>
<td>identifier</td>
</tr>
<tr>
<td>jwks_uri</td>
<td>URL of the Json Web Key Set</td>
<td>/jwks</td>
</tr>
<tr>
<td>request_parameter_supported</td>
<td>Boolean value specifying whether the OP supports use of the request parameter</td>
<td>true</td>
</tr>
<tr>
<td>request_uri_parameter_supported</td>
<td>Boolean value specifying whether the OP supports use of the request_uri parameter</td>
<td>true</td>
</tr>
<tr>
<td>require_request_uri registered</td>
<td>Boolean value specifying whether the OP requires any request_uri used to be pre-registered</td>
<td>false</td>
</tr>
<tr>
<td>response_modes_supported</td>
<td>JSON array containing a list of response_mode values</td>
<td>[query, fragment]</td>
</tr>
<tr>
<td>revocation_endpoint</td>
<td>URL used to notify the authorization server that a refresh or access token is not longer valid</td>
<td>NOT SUPPORTED</td>
</tr>
<tr>
<td>scopes_supported</td>
<td>JSON array containing a list of scopes</td>
<td>[address, email, openid, phone, profile, role]</td>
</tr>
<tr>
<td>subject_types_supported</td>
<td>JSON array containing a list of subject identifier types</td>
<td>[public]</td>
</tr>
<tr>
<td>token_endpoint</td>
<td>URL of the token endpoint</td>
<td>/token</td>
</tr>
<tr>
<td>token_endpoint_auth_methods_supported</td>
<td>JSON array containing a list of Client authentication methods supported by this Token Endpoint</td>
<td>[client_secret_basic, client_secret_post, client_secret_jwt, private_key_jwt]</td>
</tr>
<tr>
<td>userinfo_endpoint</td>
<td>URL of the userinfo endpoint</td>
<td>/userinfo</td>
</tr>
<tr>
<td>version</td>
<td>Version of the contract</td>
<td>1.0</td>
</tr>
<tr>
<td>registration_endpoint</td>
<td>URL of the dynamic client registration endpoint</td>
<td>/registration</td>
</tr>
</tbody>
</table>
3.5 Json Web Keys

TO COMPLETE

3.6 Registration

TO COMPLETE

3.7 Introspection

TO COMPLETE
Quick start

4.1 In 5 steps

The framework can “easily” be installed on any ASP.NET 5 project.

4.1.1 Nuget feeds

Add two new feeds into your Nuget’s configuration file
& https://www.myget.org/F/thabart/api/v3/index.json

Note : The default file location is : %APPDATA%Nuget\Nuget.config

4.1.2 Your environment

Before you can start using the framework. Be-sure that all the pre-requisistes are installed on your machine otherwise you’ll not be able to develop an ASP.NET 5 project.

• If the .NET version manager (DNVM) is not installed then open a command dos window and execute the powershell command.
• Upgrade the DNX tool to the latest version (the current one is : 1.0.0-rc1-update1).
• ASP.NET 5 project can be developed with Visual Studio 2015 or an open-source IDE on which you can install OmniSharp for example Visual Studio Code, ATOM, Brackets etc...

Commands:
dnvm upgrade

4.1.3 Installation

Add the Nuget package “SimpleIdentityServer.Host - 1.0.0-rc1-final” into your project and restore all the dependencies :

dnu restore
4.1.4 Startup

The request pipeline can be configured in the Startup class, it is used to handle all requests made to the application. It provides the entry point for an application, and is required for all applications. The startup class must define a Configure method, and may optionally also define a ConfigureServices.

Configure method

The Configure method is used by the developer to register a set of middlewares which will interact with the HTTP request / response pipeline, for example: the extension method UseCookieAuthentication adds a new middleware which is trying to authenticate the incoming HTTP requests. Use the extension method UseSimpleIdentityServer to add Simple Identity Server into your ASP.NET 5 hosted project. The snippet code below configures an In Memory identity server instance:

```csharp
services.AddSimpleIdentityServer(new DataSourceOptions {
    DataSourceType = DataSourceTypes.InMemory,
    ConnectionString = connectionString,
    Clients = Clients.Get(),
    JsonWebKeys = JsonWebKeys.Get(),
    ResourceOwners = ResourceOwners.Get(),
    Scopes = Scopes.Get(),
    Translations = Translations.Get()
}, _swaggerOptions);
```

In the above example all open-id assets are stored into memory. They can also be persisted into an SqlServer database. For more information about the configuration please refer to the next part.

ConfigureServices method

The ConfigureServices method is called before Configure. It configures and registers services that are used by your application. Simple Identity Server cannot work if its dependencies are not registered. Call the method AddSimpleIdentityServer to register all the dependencies:

```csharp
app.UseSimpleIdentityServer(new HostingOptions
{
    IsDeveloperModeEnabled = false,
    IsMicrosoftAuthenticationEnabled = true,
    MicrosoftClientId = Configuration["Microsoft:ClientId"],
    MicrosoftClientSecret = Configuration["Microsoft:ClientSecret"],
    IsFacebookAuthenticationEnabled = true,
    FacebookClientId = Configuration["Facebook:ClientId"],
    FacebookClientSecret = Configuration["Facebook:ClientSecret"]
}, _swaggerOptions);
```

For more details about the properties, please refer to the next part.

SwaggerOptions

You’ve probably noticed that SwaggerOptions is passed to the procedures UseSimpleIdentityServer and AddSimpleIdentityServer. It should be used only if you desired to interact with the different OpenId end-points via Swagger. It’s preferable to disable it in a product environment.

```csharp
 SwaggerOptions _swaggerOptions = new SwaggerOptions
{
```

Chapter 4. Quick start
Simple identity server documentation, Release 1.0

IsSwaggerEnabled = true

4.1.5 Run

Open a command prompt, navigate to your project and execute the following command:

dnx web

At the end you should be able to navigate to the home screen:

The sample project can be found here: https://github.com/thabart/SimpleIdentityServer/tree/master/SimpleIdentityServer/src/SimpleIdentityServer.Startup

4.2 Options

4.2.1 DataSourceOptions

<table>
<thead>
<tr>
<th>Key</th>
<th>Value(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSourceType</td>
<td>InMemory</td>
<td>Store open-id assets into memory</td>
</tr>
<tr>
<td></td>
<td>SqlServer</td>
<td>Store open-id assets into SqlServer database</td>
</tr>
<tr>
<td>ConnectionString</td>
<td>Connection string</td>
<td>Used to connect to SqlServer database</td>
</tr>
<tr>
<td>Clients</td>
<td>List&lt;Client&gt;</td>
<td>List of clients stored in memory</td>
</tr>
<tr>
<td>JsonWebKeys</td>
<td>List&lt;JsonWebKey&gt;</td>
<td>List of json web keys stored in memory</td>
</tr>
<tr>
<td>ResourceOwners</td>
<td>List&lt;ResourceOwner&gt;</td>
<td>List of resource owners stored in memory</td>
</tr>
<tr>
<td>Scopes</td>
<td>List&lt;Scope&gt;</td>
<td>List of scopes stored in memory</td>
</tr>
<tr>
<td>Translations</td>
<td>List&lt;Translation&gt;</td>
<td>List of translations stored in memory</td>
</tr>
</tbody>
</table>

4.2.2 HostingOptions

<table>
<thead>
<tr>
<th>Key</th>
<th>Value(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsDeveloperModeEnabled</td>
<td>string</td>
<td>• true : exception stack trace is displayed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• false : well open-id formatted exception is returned (TokenErrorResponse)</td>
</tr>
<tr>
<td>IsDataMigrated</td>
<td>bool</td>
<td>• true : mandatories open-id assets (open-id scope, claim role ...) are inserted into SqlServer database</td>
</tr>
<tr>
<td>IsMicrosoftAuthenticationEnabled</td>
<td>bool</td>
<td>Enable authentication with Microsoft account</td>
</tr>
<tr>
<td>MicrosoftClientId</td>
<td>string</td>
<td>Microsoft client id</td>
</tr>
<tr>
<td>MicrosoftClientSecret</td>
<td>string</td>
<td>Microsoft client secret</td>
</tr>
<tr>
<td>IsFacebookAuthenticationEnabled</td>
<td>bool</td>
<td>Enable authentication with Facebook account</td>
</tr>
<tr>
<td>FacebookClientId</td>
<td>string</td>
<td>Facebook client id</td>
</tr>
<tr>
<td>FacebookClientSecret</td>
<td>string</td>
<td>Facebook client secret</td>
</tr>
</tbody>
</table>

4.2. Options
We are using docker to easily deploy and start the application inside a docker container. There’re four components deployed:

2. SimpleIdentityServerManagerApi: Web.Api service used to manage the open-id assets.
3. SimpleIdentityServerManager: Angularjs website used to manage the open-id assets.
4. Database: Open-Id assets are stored for now in a SQLite3 database.

It’s pretty easy to deploy & run the application, in a command prompt execute the command below. If you’re working on Windows be-sure that your environment is correctly set-up. Armen Shirmoon explains in a very well written blog post, how to easily run Docker on Windows. If your Windows environment is not ready I invit you to read this article.

```
docker run -t -d -p 5000:5000 identitycontrib/identityserver
```

Once the docker container has been started, browse the URL: http://localhost:5000 to access to SimpleIdentityServer.