Flowpack Single sign-on Server Documentation

Release 1.0.0

Author name(s)

July 11, 2016

1 Contents **3**
- 1.1 Overview . . . 3
- 1.2 Getting started . . . 6
- 1.3 Single sign-on server . . . 12
- 1.4 Single sign-on client . . . 21
- 1.5 Development . . . 28

The Flowpack Single sign-on packages provide a distributed authentication and authorization solution for TYPO3 Flow applications. It is based on the Flow security framework and makes no special assumptions about the actual authentication method, source of account data and the authorization data that is exchanged between the systems.

CHAPTER 1

Contents

1.1 Overview

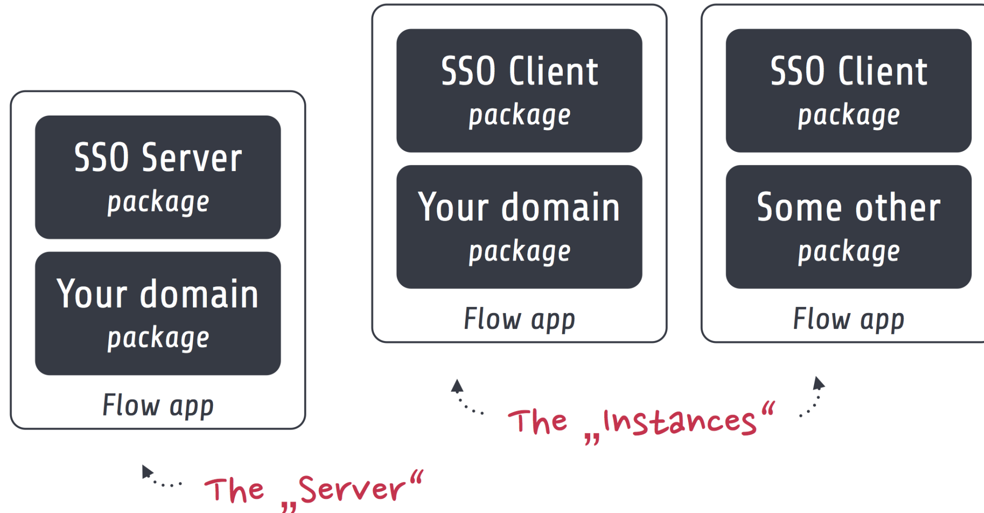
The Flowpack Single sign-on is designed for integration in TYPO3 Flow applications and offers a full single sign-on solution without the need of external components. The Server package can be used to build a custom authentication server based on TYPO3 Flow while the Client package can be used to integrate existing TYPO3 Flow applications into the single sign-on.

We designed the solution for ease of use and a seamless authentication experience for the user.

1.1.1 Features

- Easy integration into existing TYPO3 Flow applications
- Flow security framework integration, re-use of existing authentication providers (e.g. *LDAP*, *UsernamePassword*, *OpenID*)
- Flexible account data mapping (transfer custom properties of parties)
- Session expiration synchronization
- Remote session management capabilities
- Single Sign-off
- Account switching (impersonate)
- Sessions can use existing Flow cache backends (*Redis*, *Memcache*, *APC*)
- RSA signing of server-side requests

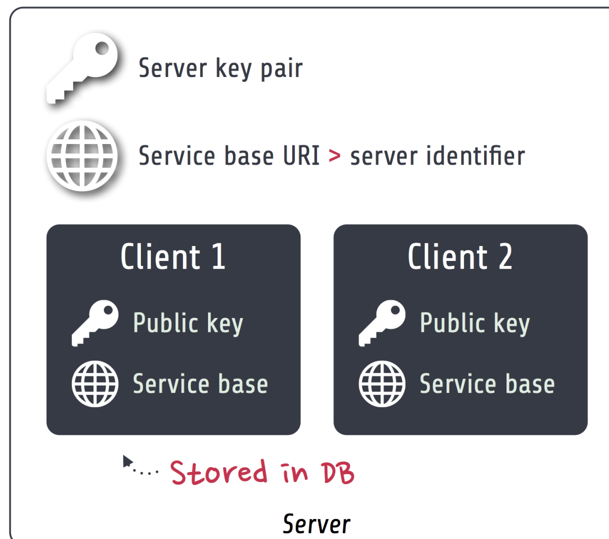
1.1.2 Architecture



The architecture is designed to be *highly extensible* and *fully integratable* in an existing TYPO3 Flow application.

Server

A Single sign-on server is a TYPO3 Flow application that provides a central authentication system which is accessed by the instances. The server consists of the Server package and a domain package that implements a party model for the authentication and provides possible extensions to the Single sign-on data exchange.



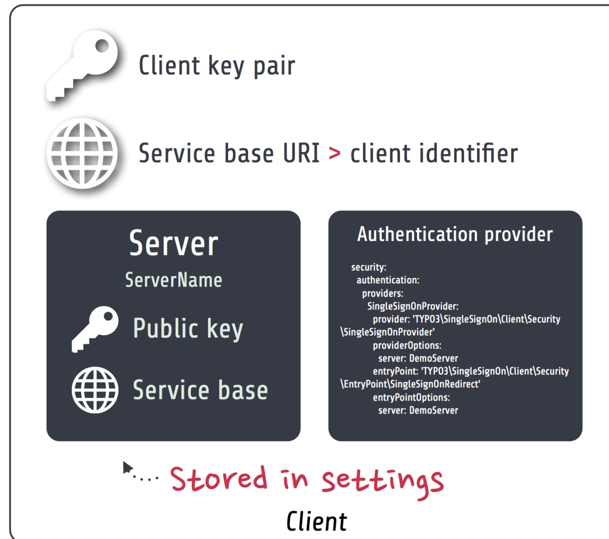
The server has a *public / private key pair* and exports HTTP service as the *Service base URI* (e.g. *http://ssoserver.local/sso/*). The service base URI is also used as the unique server identifier.

All the instances have to be registered as a single sign-on client with their public key and service base URI. This allows for (signed) server-side requests initiated by the client or the server. The client public key restricts access to the single sign-on only to explicitly registered clients. The clients are persisted as entities inside a configured database. A management interface for the clients can be implemented in a custom package.

See Single sign-on server for more information about implementing a custom server application.

Instance

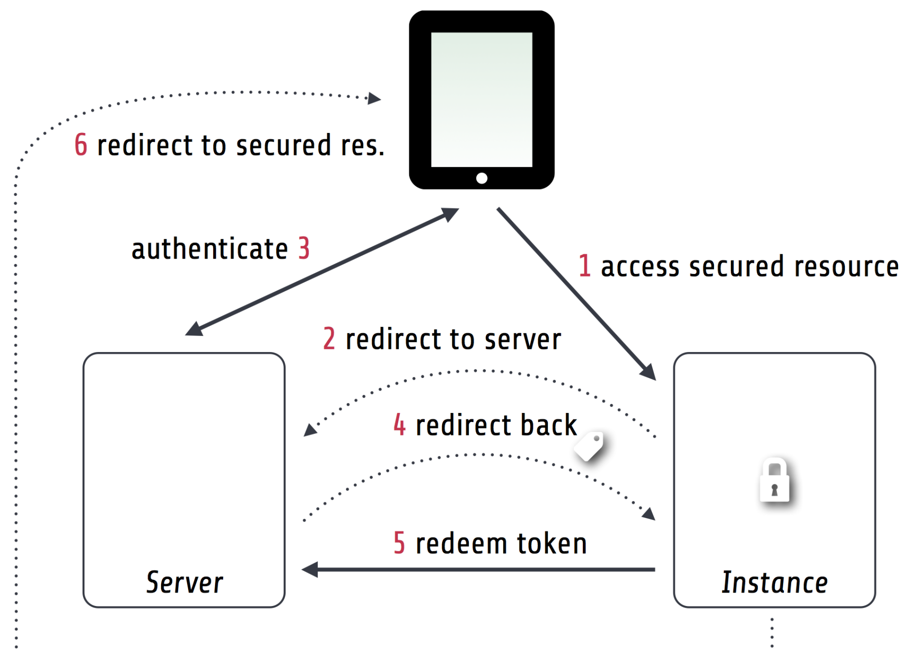
An instance is a (TYPO3 Flow) application that utilizes a Single sign-on server for authentication using the Client package. The Flowpack Single sign-on can be one of multiple authentication methods on the instance. Usually there will be a larger number of instances in a typical setup.



The single sign-on client on the instance has a public / private key pair and a *Service base URI* as a unique client identifier. The client needs at least one configured single sign-on server with the server public key and service base URI. The client is used by the instance through the Flow security framework as a special authentication provider.

1.1.3 Authentication round trip

This is a simple round trip for access to a secured resource on an instance without prior authentication:



1. A user accesses a secured resource on an instance

2. Since no account is authenticated on the instance the user is redirected to a configured server
3. The user will authenticate on the server through a configured authentication provider (e.g. username / password)
4. The server redirects back to the instance and passes an encrypted access token
5. The instance decrypts the access token and does a server-side request to redeem the token on the server, the server verifies the token and returns the account data and authorization information (e.g. roles)
6. The instance authenticates an account locally and redirects to the original secured resource

1.2 Getting started

Tip: Before building a custom single sign-on server and integration of the client in existing Flow applications we recommend to have a look at a the demo setup.

We provide TYPO3 Flow demo applications for both the server and an exemplary instance. To make the setup easier we also provide the demo as a Vagrant box (a tool for a development environment in virtual machines).

Warning: Do not use the *Flowpack.SingleSignOn.DemoServer* package in production! It contains code that is meant for testing and allows creation of users and session management over an unsecured HTTP API.

The following demo credentials are available on the server:

Username	Password	Role
admin	password	Administrator
user1	password	User
user2	password	User

1.2.1 Setting up the Vagrant demo

First clone the DemoVagrant repository that contains the Vagrant box setup:

```
$ git clone https://github.com/Flowpack/Flowpack.SingleSignOn.Demo-Vagrant.git DemoVagrant
$ cd DemoVagrant
```

Then install Vagrant for your operating system, install the *librarian* gem for downloading bundled cookbooks and start the vagrant box:

```
$ gem install librarian
$ librarian-chef install
  ...
$ vagrant up
```

The virtual machine should now boot and start to provision the demo setup (this can take a while).

Set up host entries in your */etc/hosts* (or similar file, depending on your operating system):

```
10.11.12.23 ssodemoserver.vagrant
10.11.12.23 ssodemoinstance.vagrant ssodemoinstance2.vagrant
```

Browse to http://ssodemoserver.vagrant/ and you should see the demo server frontend. A second instance is available on http://ssodemoserver2.vagrant/ for running multi-instance acceptance tests.

1.2.2 Manually setting up the demo server and instance

The demo setup consists of a demo server and a demo instance bundled in two TYPO3 Flow distributions. You should follow the steps in the TYPO3 Flow Quickstart for a general setup for Flow development if not yet done.

Each distribution should be cloned into a separate directory:

```
$ mkdir singlesignon-demo
$ cd singlesignon-demo
```

Setting up the server

Clone the repository, install dependencies with Composer:

```
$ git clone https://github.com/Flowpack/Flowpack.SingleSignOn.DemoServer-Distribution.git DemoServer
$ cd DemoServer
$ path/to/composer.phar install --dev
```

Create a *Configuration/Settings.yaml*:

```
TYPO3:
  Flow:
    persistence:
      backendOptions:
        dbname: ssodemoserver # Create this database
        host: localhost
        user: root    # Fill in username
        password: '' # Fill in password

Flowpack:
  SingleSignOn:
    Server:
      server:
        serviceBaseUri: 'http://ssodemoserver.local/sso/'
        publicKeyFingerprint: ''

    DemoServer:
      demoInstanceUri: 'http://ssodemoinstance.local/'
      clients:
        -
          serviceBaseUri: 'http://ssodemoinstance.local/sso/'
```

Run migrations and demo setup:

```
$ ./flow doctrine:migrate
$ ./flow flowpack.singlesignon.demoserver:demo:setup
```

Setting up the instance

Clone the repository, install dependencies with Composer:

```
$ git clone https://github.com/Flowpack/Flowpack.SingleSignOn.DemoInstance-Distribution.git DemoInsta
$ cd DemoInstance
$ path/to/composer.phar install --dev
```

Create a *Configuration/Settings.yaml*:

```
TYPO3:
  Flow:
    persistence:
      backendOptions:
        dbname: ssodemoinstance # Create this database
        host: localhost
        user: root    # Fill in username
        password: '' # Fill in password

Flowpack:
  SingleSignOn:
    Client:
      client:
        serviceBaseUri: 'http://ssodemoinstance.local/sso/'
        publicKeyFingerprint: ''
      server:
        DemoServer:
          serviceBaseUri: 'http://ssodemoserver.local/sso/'
          publicKeyFingerprint: ''

    DemoInstance:
      demoServerUri: 'http://ssodemoserver.local/'
```

Run migrations and demo setup:

```
$ ./flow doctrine:migrate
$ ./flow flowpack.singlesignon.demoinstance:demo:setup
```

You should create a virtual host configuration for both distributions. We expect the hosts *ssodemoinstance.local* and *ssodemoserver.local* for the example configuration.

After setting up everything you should be able to access http://ssodemoserver.local/ and see the demo server front page.

1.2.3 Demo walkthrough

You could test the following scenarios:

- Go to demo instance, request *secure action*: A login form on the server will be displayed. After login with one of the *demo credentials* you should be redirected back to the secure action and be authenticated on the server and instance.
- Authenticate on server, request *secure action* on instance: No login form is displayed if an authenticated session already exists and the session is transferred to the instance using redirects and server-side requests.
- Authenticate on server and instance, logout from server: When going to the instance again you should see, that the session was automatically invalidated using a server-side request.
- Authenticate on server and instance, logout on instance: When going to the server you should see, that the session was automatically invalidated using a server-side request.

About the demo server

The demo server distribution has a package *Flowpack.SingleSignOn.DemoServer* for custom domain models and extensions to the single sign-on. This package also implements a UI for demonstration and requires the *Flowpack.SingleSignOn.Server* package which does all the heavy-lifting for the single sign-on.

The *User* entity of the DemoServer is a simple *AbstractParty* implementation:

```
/**
 * @Flow\Entity
 */
class User extends AbstractParty {

    /**
     * @var string
     */
    protected $firstname = '';

    /**
     * @var string
     */
    protected $lastname = '';

    /**
     * @var string
     */
    protected $company = '';

    ...
}
```

Basically any *AbstractParty* implementation will work for the single sign-on.

The *LoginController* in the DemoServer package handles the actual authentication (on redirection from an instance or directly on the server) against a configured authentication provider and is the same as for any other Flow application:

```
class LoginController extends AbstractAuthenticationController {

    public function indexAction() {
    }

    protected function onAuthenticationSuccess(\TYPO3\Flow\Mvc\ActionRequest $originalRequest = NULL)
        if ($originalRequest !== NULL) {
            $this->redirectToRequest($originalRequest);
        }

        $this->addFlashMessage('No original SSO request present. Account authenticated on server.',
        $this->redirect('index', 'Standard');
    }

    public function logoutAction() {
        parent::logoutAction();

        $this->addFlashMessage('You have been logged out');
        $this->redirect('index', 'Standard');
    }

}
```

In the *onAuthenticationSuccess* method a check is made for an original request (which is passed from an entry point) and a flash message is displayed otherwise. The magic happens because the client package redirects the user to an *SSO authentication endpoint* where the authentication is started and a configured entry point redirects the user to the *LoginController* if no account is authenticated.

The configuration of the entry point is done like in any other Flow application:

```
TYPO3:
  Flow:
    security:
      authentication:
        providers:
          DefaultProvider:
            provider: PersistedUsernamePasswordProvider
            entryPoint: WebRedirect
            entryPointOptions:
              uri: 'login'
```

Tip: See the TYPO3 Flow security framework documentation for more information about authentication providers and entry points.

The only other relevant configuration contains the server key pair fingerprint and service base URI:

```
Flowpack:
  SingleSignOn:
    Server:
      server:
        keyPairFingerprint: bb5abb57faa122cc031e3c904db3d751
        serviceBaseUri: 'http://ssodemoserver.local/sso/'
```

The REST services of the server package have to be registered by mounting the routes in the global *Routes.yaml*:

```
-
  name: 'SingleSignOn'
  uriPattern: 'sso/<SingleSignOnSubroutes>'
  subRoutes:
    SingleSignOnSubroutes:
      package: Flowpack.SingleSignOn.Server
```

This route also defines the *service base URI* of the server, which is a mandatory configuration for all SSO clients.

For the demo setup we have provided a convenient setup command for the key creation and SSO client registration. To create a new key pair the *ssokey:generatekeypair* command can be used.

The DemoServer package contains some special controllers for demonstration purposes (*SessionsController* and *ConfigurationController*) which are not needed for the single sign-on.

About the demo instance

The demo instance distribution also has a package *Flowpack.SingleSignOn.DemoInstance* which implements a demo UI and configures the single sign-on as a Flow authentication provider. The *secure action* is implemented by restricting access to a controller action in the *Policy.yaml* just like in every other Flow application.

The user entity on the instance is mostly a copy of the server model but is not meant for persistance but transient usage:

```
/**
 * @Flow\Entity
 */
class User extends \TYPO3\Party\Domain\Model\AbstractParty {

    /**
     * The username of the user
     *
     * @var string
```

```
     */
    protected $username;

    /**
     * @var string
     */
    protected $firstname = '';

    /**
     * @var string
     */
    protected $lastname = '';

    /**
     * @var string
     */
    protected $company = '';

    /**
     * @var string
     */
    protected $role = '';

    ...
}
```

The single sign-on does not require a transient party model, but the *SimpleGlobalAccountMapper* that comes with the *Flowpack.SingleSignOn.Client* package does always create a fresh account instance and maps the properties of the server party to a configured type on the instance (see setting *Flowpack.SingleSignOn.Client.accountMapper.typeMapping*).

The instance uses the single sign-on by configuring the authentication provider *SingleSignOnProvider* in its *Settings.yaml*:

```
TYPO3:
  Flow:
    security:
      authentication:
        providers:
          SingleSignOnProvider:
            provider: 'Flowpack\SingleSignOn\Client\Security\SingleSignOnProvider'
            providerOptions:
              server: DemoServer
              globalSessionTouchInterval: 5
            entryPoint: 'Flowpack\SingleSignOn\Client\Security\EntryPoint\SingleSignOnRedirect'
            entryPointOptions:
              server: DemoServer
```

This configures an authentication provider with name *SingleSignOnProvider* to use the *SingleSignOnProvider* from the single sign-on client package. It's important to also configure the entry point when using the single sign-on provider. The entry point will redirect the user to the server if no session is authenticated locally and handles the parameter passing.

The provider and entry point options refer to a server by an identifier *DemoServer*. This identifier is configured in the *Flowpack.SingleSignOn.Client* settings:

```
Flowpack:
  SingleSignOn:
    Client:
```

```
      client:
        serviceBaseUri: http://ssodemoinstance.dev/sso
        publicKeyFingerprint: bb45dfda9f461c22cfdd6bbb0a252d8e

      server:
        DemoServer:
          serviceBaseUri: http://ssodemoserver.dev/sso/
          publicKeyFingerprint: bb5abb57faa122cc031e3c904db3d751

      accountMapper:
        typeMapping:
          # Map a user type from the server to one of the instance, more complex scenarios
          # need a specialized account mapper implementation (see GlobalAccountMapperInterface)
          'Flowpack\SingleSignOn\DemoServer\Domain\Model\User': 'Flowpack\SingleSignOn\DemoInstance\I
```

Configuring the authentication provider and entry point tells the Flow security framework to use the single sign-on for authentication. The single sign-on client needs some more settings for the client / server public key fingerprints and the service base URIs to use for redirecting back and forth during authentication.

1.3 Single sign-on server

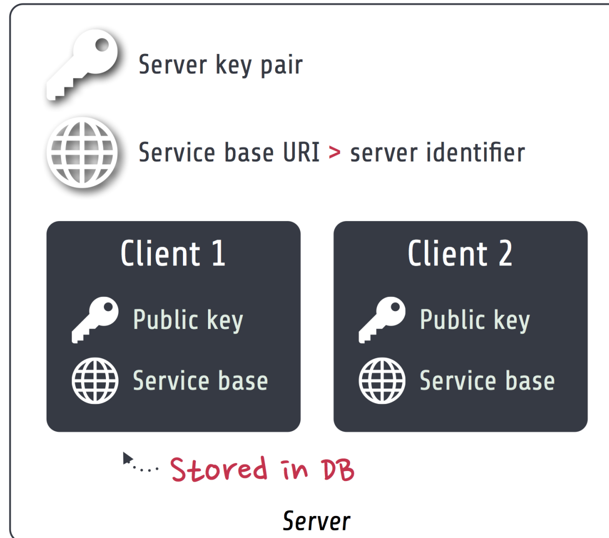
The *Flowpack.SingleSignOn.Server* package provides the components for implementing a custom single sign-on server. This package should be installed in a Flow application that implements custom domain logic and authentication configuration in a project specific package.

Requirements for a single sign-on server:

- Implementation of a Party domain model and domain logic
- Configuration of authentication
- Inclusion of routes for HTTP services
- Establish a secure server-side communication between the server and instances (e.g. SSL with certificates)
- Party and account management (optional)
- Basic user interface for login and display of messages (optional)

1.3.1 Components

This is a schematic view of the single sign-on server components:



Server key pair The server has a *public / private key pair* for encryption and verification of requests. The public key is shared with all instances that should access the server.

Service base URI The server exports HTTP services on a specific URL path. This path acts as the *Service base URI* (e.g. *http://ssoserver.local/sso/*) or *server identifier*.

Clients All the instances have to be registered as a single sign-on client with their public key and service base URI. This allows for (signed) server-side requests initiated by the client or the server. The client public key restricts access to the single sign-on only to explicitly registered clients. The clients are persisted as entities inside a configured database. A management interface for the clients can be implemented in a custom package.

Note: The server uses the default Flow security framework for authentication during single sign-on requests. So a user that doesn't have an authenticated session on the server will be delegated to one of the configured authentication providers.

TODO Show usage of authentication provider and accounts on server

1.3.2 Configuration

Package configuration

The *Flowpack.SingleSignOn.Server* package provides the following default configuration:

```
Flowpack:
  SingleSignOn:
    Server:
      server:
        # The service base URI for this server
        serviceBaseUri: ''
        # Mandatory key pair uuid (fingerprint) for the SSO server
        keyPairFingerprint: ''
      log:
        backend: TYPO3\Flow\Log\Backend\FileBackend
        backendOptions:
          logFileURL: %FLOW_PATH_DATA%Logs/SingleSignOn_Server.log
          createParentDirectories: TRUE
```

```
          severityThreshold: %LOG_WARN%
          maximumLogFileSize: 10485760
          logFilesToKeep: 1
          logMessageOrigin: FALSE

        # Enable logging of request signing (all signed requests)
        logRequestSigning: FALSE
```

Option	Description	Mandatory	Type	Default
server.serviceBaseUri	The service base URI for this server	Yes	string	
server.keyPairFingerprint	Key pair fingerprint for the server	Yes	string	
log.backend	Log backend type for the single sign-on logger	No	string	FileBackend
log.backendOptions	Log backend options for the single sign-on logger	No	array	see Settings
log.logRequestSigning	Controls logging of signed requests via an aspect (for debugging)	No	boolean	FALSE
accountMapper.configuration	Serialization of account data for client account mapping in the default *SimpleClientAccountMapper*.	No	array	NULL

Note: The package also configures some settings for TYPO3 Flow. For the signed requests a security firewall filter with the name *ssoServerSignedRequests* is configured. This filter can be modified or removed in another package configuration or global configuration.

Caches

A special cache with the identifier *Flowpack_SingleSignOn_Server_AccessToken_Storage* is used for the storage of *Access tokens*. It defaults to a *FileBackend* as the cache backend.

Routes

The routes of the server package have to be registered in the global *Routes.yaml*:

```
##
# Flowpack.SingleSignOn.Server subroutes
#
```

```
-
  name: 'SingleSignOn'
  uriPattern: 'sso/<SingleSignOnSubroutes>'
  subRoutes:
    SingleSignOnSubroutes:
      package: Flowpack.SingleSignOn.Server
```

The path *sso/* can be freely chosen but will be part of the server service base URI that needs to be configured in the client configuration on a single sign-on instance.

1.3.3 Commands

ssoserver:registerclient

The server exposes a *ssoserver:registerclient* command for client registration from the CLI:

```
Add a client

COMMAND:
  flowpack.singlesignon.server:ssoserver:registerclient

USAGE:
  ./flow ssoserver:registerclient <base uri> <public key>

ARGUMENTS:
  --base-uri           The client base URI as the client identifier
  --public-key         The public key fingerprint (has to be imported using the
                       RSA wallet service first)

DESCRIPTION:
  This command registers the specified client at the SSO server.
```

The key pair has to be created on the instance using the *ssokey:generatekeypair* command (TODO reference section in the client doc).

Example:

```
$ ./flow ssoserver:registerclient --base-uri http://ssoinstance.local/sso/ --public-key c1285a470f0f
```

ssoserver:removeexpiredaccesstokens

The *ssoserver:removeexpiredaccesstokens* command cleans up expired *access tokens* from the underlying cache back-end.

```
Remove expired access tokens

COMMAND:
  flowpack.singlesignon.server:ssoserver:removeexpiredaccesstokens

USAGE:
  ./flow ssoserver:removeexpiredaccesstokens

DESCRIPTION:
  This will remove all expired access tokens that were not redeemed from the underlying storage.
  This command should be executed in regular intervals for cleanup.
```

This command should be executed in regular intervals (e.g. using a cron task) to clean up the access token storage.

1.3.4 Logging

The server package *Configuration* configures a default logger that is used in the single sign-on package for logging various events. The default file for the logger in *Production* context is *Data/Logs/SingleSignOn_Server.log*.

Tip: The log should always be consulted if problems with the single sign-on need to be investigated. In the log level *INFO* it provides an overview of all single sign-on authentications.

Warning: Access to the logs must be restricted in production since it contains sensible information (session identifiers and access tokens).

The logger interface can be injected in any package to log into the same destination:

```
/**
 * @var \Flowpack\SingleSignOn\Server\Log\SsoLoggerInterface
 * @Flow\Inject
 */
protected $ssoLogger;
```

1.3.5 Client registration

Clients of single sign-on instances need to be registered on the server before they can participate in the single sign-on. The server needs the client service base URI and public key for encryption of request parameters, server-side client notification and request signing.

The server package implements the *ssoserver:registerclient* command for that purpose.

Programmatic client registration

For deployments with many single sign-on instance and respective clients a programmatic registration could be wanted. Since clients are represented by persisted entities this is as easy as creating a new entity and adding it to the repository.

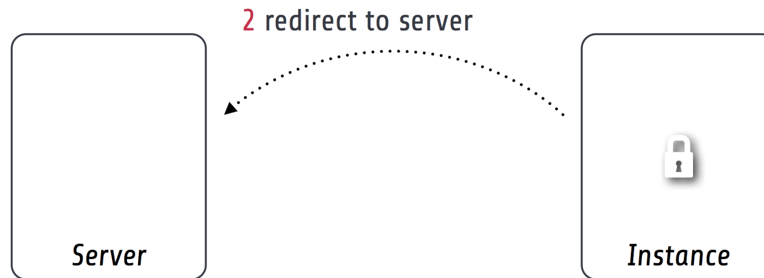
```
$ssoClient = new \Flowpack\SingleSignOn\Server\Domain\Model\SsoClient();
$ssoClient->setServiceBaseUri($baseUri);
$ssoClient->setPublicKey($publicKeyFingerprint);
$this->ssoClientRepository->add($ssoClient);
```

It should be trivial to build a management interface for client management.

Tip: Programmatic access to the RSA wallet for key management is easy with an instance of *TYPO3FlowSecurityCryptographyRsaWalletServiceInterface*.

1.3.6 Authentication endpoint

The single sign-on server exposes a public controller action for handling authentication requests from clients. This is called the *authentication endpoint*.



/sso/authentication?callbackUri=...&ssoClientIdentifier=...&signature=...

The single sign-on client entry point on the instance will redirect a user to the *authentication endpoint* on the server if no authenticated session is present locally on the client. This redirect happens *in the browser* of the user to access an existing web session of the server (if the user authenticated on the server or another instance before).

To secure the parameters and guard against a possible redirection attack the parameters are signed by the client in the *signature* query argument. This is done using the private key of the client, so the server can verify the signature with the client public key and also verify the identity of the request.

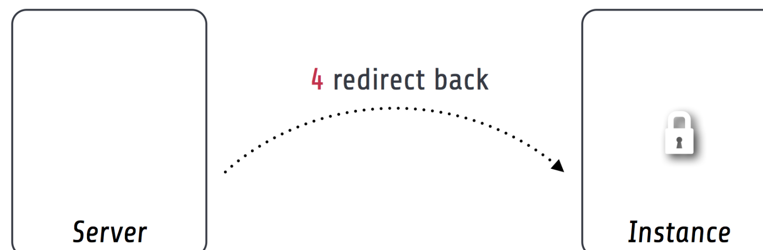
A *ClientNotFoundException* will be thrown if the client was not registered on the server.

The authentication endpoint action will act as an authentication controller and eventually call *$this->authenticationManager->authenticate()*, so the user has to authenticate using a configured authentication provider to proceed. This approach integrates nicely with the Flow security framework and allows the usage of all available authentication providers and entry points.

Note: There has to be an entry point (e.g. *WebRedirect*) that matches the authentication endpoint request. Otherwise a user would not have a chance to authenticate using a login form (or other means depending on the authentication provider). See *About the demo server* for an example configuration.

1.3.7 Authentication callback

After the user was authenticated on the server or if the user was already authenticated an *access token* is created on the server and sent to the instance via an redirect *in the browser* to the client *authentication callback*.



/sso/authentication/callback?callbackUri=...&__accessToken=...&__signature=...

Since the redirect to the client should be considered insecure the access token is encrypted and signed with the client public key and server private key.

Access tokens

The access token stores the account that was authenticated on the server and the client that initiated the authentication request. An access token is identified by a string of 32 random characters. An access token has an expiry setting which defaults to 60 seconds. This and the fact that an access token is deleted after redemption should prevent replay attacks.

Access tokens are stored in a cache backend *Flowpack_SingleSignOn_Server_AccessToken_Storage*. The default *configuration* uses a *FileBackend*. The cache backend allows for a flexible and lightweight storage of access tokens with automatic expiration and garbage collection.

The server package provides a *ssoserver:removeexpiredaccesstokens* command for the maintenance of the cache backend that will remove expired access tokens that were not redeemed. This command should be executed in regular intervals for garbage collection of the cache backend.

1.3.8 Access token redemption

After the single sign-on client has verified the access token a server-side **'signed request'_** is made to exchange the access token for the actual account data and *single sign-on session identifier*. This measure also prevents injection of arbitrary account data into the callback URI by breaking the signature.

POST /sso/token/jNkmy06oC1gm4xozKt1FR579/redeem

With a valid access token the server will:

- get the original session identifier and account from the access token
- invalidate (remove) the access token
- register the single sign-on client in the session for *Client notification*
- perform *Account mapping* to transform the server account into authentication and authorization information for the client
- respond with a JSON representation of the mapped account and the server session identifier

The client will transform the returned account data into a local account (persistent or transient) using a *global account mapper* and authenticate this account locally.

Note: The redeem access token request is not public and is guarded by a signed request filter by default. Additional measures to secure this channel should be installed in production environments.

1.3.9 Account mapping

Since the account and party information that is needed on an instance is dependent on the actual requirements of an application the single sign-on solution does not impose a fixed schema for the information.

With an implementation of the *ClientAccountMapperInterface* any strategy for a transformation given the account and client instance can be implemented:

```
interface ClientAccountMapperInterface {

    /**
     * Map the given account as account data for an instance
     *
     * @param \Flowpack\SingleSignOn\Server\Domain\Model\SsoClient $ssoClient
     * @param \TYPO3\Flow\Security\Account $account
     * @return array
     */
    public function getAccountData(
            \Flowpack\SingleSignOn\Server\Domain\Model\SsoClient $ssoClient,
            \TYPO3\Flow\Security\Account $account
    );

}
```

The *getAccountData* method has to return the serialized account information based on the given single sign-on client and account object. It is possible to differentiate between clients and return different account information depending on the client with this approach.

SimpleClientAccountMapper

A basic implementation of a client account mapper is included in the server package with the *SimpleClientAccountMapper* class and will be used by default.

Example account data:

```
accountIdentifier: 'jdoe'
roles: ['Vendor.MyPackage:User']
party:
  __type: 'Vendor\MyPackage\ExampleParty'
  company: 'Acme Inc.'
```

The *accountIdentifier* and *roles* keys are always returned and do not need any configuration.

The implementation will serialize the party properties according to the *configuration* property which is configurable via the *Flowpack.SingleSignOn.Server.accountMapper.configuration* setting.

The default configuration will handle the party type *Person* and returns all simple properties including the name:

```
array(
    'party' => array(
            '_exposeType' => TRUE,
            '_descend' => array('name' => array())
    )
);
```

For any other party implementation it will just return accessible properties directly under the party object, so for relational party data a custom configuration has to be given.

It is important that the type of the party is exposed as the key *__type* for the default implementation of the **'global account mapper'_** on the client (class *SimpleGlobalAccountMapper*).

Note: The exchange of account data is deliberately unconstrained to allow for a fully flexible exchange of data. But the implementation of the *ClientAccountMapperInterface* on the server and *GlobalAccountMapperInterface* on the

client have to match in terms of the exported and expecteded properties.

1.3.10 Client notification

The client notification is used to destroy sessions remotely by a server-side request to the client. This is mainly used for synchronized logout (*Single sign-off*) and account switching on the server.

The server declares a *SsoClientNotifierInterface* interface for this purpose and provides two implementations using a synchronous (*SimpleSsoClientNotifier*) and parallel (*ParallelSsoClientNotifier*) strategy for the HTTP requests. The *SimpleSsoClientNotifier* is the default implementation configured in the server package *Objects.yaml*. In scenarios that register a lot of instances for one session the *ParallelSsoClientNotifier* can reduce the latency on logout or account switching by using parallel HTTP requests with a multi-threading engine.

A destroyed session on the client will require authentication through the single sign-on mechanism on the next request to a secured resource on the client. This ensures an updated authentication state on the instance.

Note: The client notification will destroy all session data on the client. If the instance stores important data in the session this data will be lost on logout or account switching on another instance or the server.

1.3.11 Session synchronization

The TYPO3 Flow session has a configurable interval for inactivity that is used to expire sessions after a certain time of inactivity on the next access or through garbage collection.

In a single sign-on scenario we have to consider multiple Flow sessions (after authentication with at least one instance):

- One *global* session on the single sign-on server
- One ore more *local* sessions on the instances

The server and instances could have different inactivity timeouts configured for the Flow session which leads to an effect where the user is still authenticated on the client but the server session is already expired due to inactivity (for most scenarios the user will access the server very infrequently). It is desirable that the session lifetime is synchronized in a single sign-on setup, such that an expired session on the server will also expire the session on the instances.

The Flowpack single sign-on solution does use a regular *touch* on the *global* session from the client through a special server-side signed request. The interval and frequency is configurable for the single sign-on client.

The server will respond with an error code *SessionNotFound* if the session was not found / inactive and the client will mark the authentication token as no longer authenticated.

1.3.12 Account impersonation

The *authentication endpoint* gets the current account that should be passed to the instance through the *AccountManager* service, which is implemented in the server package.

The method *impersonateAccount* allows to *impersonate* another account that will be visible as the globally authenticated account. The original account is still authenticated on the server which allows to switch back to the original or yet another account. As in the case of re-authentication on the server all registered client sessions are destroyed on impersonation.

This feature could be used to implement multi-tenant applications where one global account is able to use multiple other accounts and the user should be able to select the currently active account.

Note: A single sign-on server UI should always use the methods in *AccountManager* to get the currently active account (through *getServerAccount* or *getImpersonatedAccount*) to display authentication information.

1.3.13 HTTP services

This is a list of all HTTP services (controller actions) that are exposed by the server. The URI path depends on the global Routes.yaml that mounts the package subroutes, we expect the routes to be mounted at */sso/<SingleSignOnSubroutes>*.

Public

/sso/authentication Route for the *authentication endpoint*, has to be accessible for all users that should authenticate using the single sign-on.

Private

The controller for these routes are protected by a signed request firewall filter and should only be accessible by instances. We strongly suggest to take additional measures for securing the server-side channel between the server and instances (e.g. SSL with client certificates, firewall rules, additional request filter).

Warning: The default TYPO3 Flow routes could allow access to controller actions even though the URI paths are secured by a firewall or webserver configuration.

/sso/token/{accessToken}/redeem Route for the *access token redemption*, is used by the single sign-on client to verify the access token and to exchange it for account data and the global session identifier.

/sso/session/{sessionId}/touch Route for the *session synchronization* by allowing a client to touch the global session in regular intervals and get feedback about the session status.

/sso/session/{sessionId}/destroy Route for the *single sign-off* to destroy the global session when a user logs out on an instance.

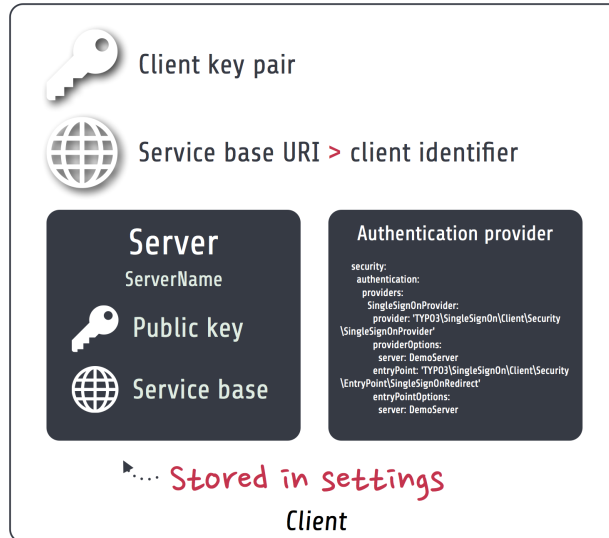
1.4 Single sign-on client

The *Flowpack.SingleSignOn.Client* package is needed on any instance that should participate in the single sign-on process. It can be installed in any TYPO3 Flow (> version 2.x) application and configured through the security framework.

No special requirements exist for an application to become a single sign-on instance, except that it should be reachable by the server for the server-side requests (e.g. for *Client notification*). Furthermore the client does not require any persistence storage (besides the default Flow session) or database access.

1.4.1 Components

This is a schematic view of the single sign-on client components that are part of a TYPO Flow application which is called an *instance*.



Client key pair The client has a *public / private key pair* for encryption and request verification. The public key is shared with all servers that are used by the client (most of the time there will be only one server).

Service base URI The client exports HTTP services on a specific URL path. This path acts as the *Service base URI* (e.g. *http://ssoinstance.local/sso/*) or *client identifier*. It is used to register the client on the server.

Servers The client configuration has a list of servers that could be used for the actual authentication provider options. From an architecture point of view it's possible to use different single sign-on servers in the same application, although that should be a rare use-case.

Authentication provider and entry point The client package provides a special *SingleSignOnProvider* authentication provider and *SingleSignOnRedirect* entry point which have to be configured in the TYPO3 Flow security framework for the single sign-on to be used as the authentication method. See *Configuration / Authentication*.

1.4.2 Installation

Installation of the single sign-on client package should be done via Composer:

```
$ path/to/composer.phar require flowpack/singlesignon-client
```

Note: If you extend the client package or want to provide configuration in a custom package you should add the *flowpack/singlesignon-client* composer package as a dependency of that package for correct initialization order.

1.4.3 Configuration

Package configuration

The *Flowpack.SingleSignOn.Client* package provides the following default configuration:

```
Flowpack:
  SingleSignOn:
    Client:
      # SSO client configuration
      client:
        # The client service base URI as the client identifier
```

```
    # Must point to the URI where the SSO client routes are mounted
    serviceBaseUri: ''
    # The client key pair fingerprint
    publicKeyFingerprint: ''

  # A list of named SSO server configurations
  server:
    # Example server configuration
    #
    #   DemoServer:
    #     publicKeyFingerprint: 'bb5abb57faa122cc031e3c904db3d751'
    #     serviceBaseUri: 'http://ssoserver/sso'

  accountMapper:
    # Map a party type from the server to the instance, more complex scenarios
    # need a specialized account mapper implementation (see GlobalAccountMapperInterface)
    #
    # typeMapping:
    #   'Vendor\MyServer\Domain\Model\SomeParty': 'Vendor\MyApplication\Domain\Model\OtherParty'
    #
    typeMapping: []

  log:
    # Enable logging of failed signed requests (signature verification errors)
    logFailedSignedRequests: FALSE
```

Option	Description	Mandatory	Type	Default
client.serviceBaseUri	The service base URI for this client	Yes	string	
client.publicKeyFingerprint	Key pair fingerprint for the client	Yes	string	
server	Array of named server configurations, server name (not identifier) as the key	Yes	array	
server.ServerName.publicKeyFingerprint	Public key fingerprint of the server	Yes	string	
server.ServerName.serviceBaseUri	Service base URI of the server	Yes	string	
accountMapper.typeMapping	Party type mapping from server to client for SimpleGlobalAccountMapper	No	array	
log.logFailedSignedRequests	Controls logging of failed signed requests via an aspect (for debugging)	No	boolean	FALSE

Note: The package also configures some settings for TYPO3 Flow. For the signed requests a security firewall filter with the name *ssoClientSignedRequests* is configured. This filter can be modified or removed in another package

configuration or global configuration.

Authentication

The client has to be configured as an authentication provider on the instance to use a server for the single sign-on.

```
TYPO3:
  Flow:
    security:
      authentication:
        providers:
          SingleSignOnProvider:
            provider: 'Flowpack\SingleSignOn\Client\Security\SingleSignOnProvider'
            providerOptions:
              server: DemoServer
              globalSessionTouchInterval: 60
            entryPoint: 'Flowpack\SingleSignOn\Client\Security\EntryPoint\SingleSignOnRedirect'
            entryPointOptions:
              server: DemoServer
```

This example configuration configures an authentication provider with name *SingleSignOnProvider* (this can be chosen freely) that uses a single sign-on server configured in *Flowpack.SingleSignOn.Client.server.DemoServer*. The entry point *SingleSignOnRedirect* needs to be registered for the single sign-on to intercept unauthenticated requests to secured resources (e.g. policy restriction of a controller action) and continue after the session is transferred from the server.

The *globalSessionTouchInterval* is a provider level option that configures the amount of seconds that can pass without touching the *global session* on the server (see *Session synchronization*).

Routes

The routes of the client package have to be registered in the global *Routes.yaml*:

```
##
# Flowpack.SingleSignOn.Client subroutes
#

-
  name: 'SingleSignOn'
  uriPattern: 'sso/<SingleSignOnSubroutes>'
  subRoutes:
    SingleSignOnSubroutes:
      package: Flowpack.SingleSignOn.Client
```

The path *sso/* can be freely chosen but will be part of the client service base URI that needs to be used for *Client registration*.

1.4.4 Commands

ssokey:generatekeypair

The client package provides a *ssokey:generatekeypair* command to create a new public / private key pair (usable for client or server):

```
Generate key pair command

COMMAND:
  flowpack.singlesignon.client:ssokey:generatekeypair

USAGE:
  ./flow ssokey:generatekeypair

DESCRIPTION:
  Creates a new key pair and imports it into the wallet.
  Used by SSO client and server.
```

Example:

```
$ ./flow ssokey:generatekeypair
Created key with fingerprint: ee60cb20fab84db9136903e107657b7f
```

The returned hash is the public key fingerprint of the created key pair.

Note: The generated private key is unencrypted and should be kept secretly (the keys are stored in the RsaWallet by default in *Data/Persistent/RsaWalletData*).

ssokey:exportpublickey

The server and client need the full public key from either side. This can be done by exporting the public from a key pair on the server or instance and importing it on the other side by using the core command *security:importpublickey*. For the export of a public key the client package ships a *ssokey:exportpublickey* command:

```
Export a public key

COMMAND:
  flowpack.singlesignon.client:ssokey:exportpublickey

USAGE:
  ./flow ssokey:exportpublickey <public key fingerprint>

ARGUMENTS:
  --public-key-fingerprint
```

Example:

Generating a key pair for the single sign-on client on an instance:

```
$ cd path/to/instance

$ ./flow ssokey:generatekeypair
Created key with fingerprint: ee60cb20fab84db9136903e107657b7f

$ ./flow ssokey:exportpublickey ee60cb20fab84db9136903e107657b7f > instance.pub
```

Importing the key for the client on the server (needs file instance.pub on the server):

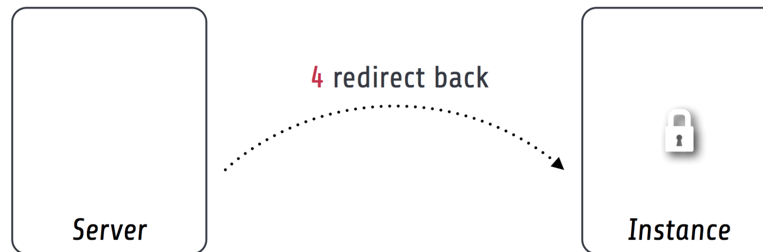
```
$ cd path/to/server

$ ./flow security:importpublickey < instance.pub
The public key has been successfully imported. Use the following uuid to refer to it in the RSAWallet
```

```
ee60cb20fab84db9136903e107657b7f
```

1.4.5 Authentication callback

The client exposes a public action for *authentication callback* that accepts a request from the server after the authentication on the server for the single sign-on was successful. The request contains query arguments for the *callbackUri*, a server generated encrypted *Access token* and a signature over the arguments for verification.



/sso/authentication/callback?callbackUri=...&__accessToken=...&__signature=...

To verify the authenticity of the request the signature is checked against the public key of the configured server and the access token is decrypted with the client private key. The next step is the *Access token redemption* with a server-side signed request from the instance to the server.

1.4.6 Account mapping

The server will respond with the account data of the authenticated account in the *global session* that is prepared through account mapping on the server.

POST /sso/token/jNkmy06oC1gm4xozKt1FR579/redeem

The client needs to map this data to a local account with a *GlobalAccountMapperInterface* implementation:

```
interface GlobalAccountMapperInterface {

    /**
     * @param \Flowpack\SingleSignOn\Client\Domain\Model\SsoClient $ssoClient
     * @param array $globalAccountData
     * @return \TYPO3\Flow\Security\Account
     */
    public function getAccount(
            \Flowpack\SingleSignOn\Client\Domain\Model\SsoClient $ssoClient,
            array $globalAccountData
    );
```

```
}
```

A default implementation of this interface is shipped in the client package with the class *SimpleGlobalAccountMapper* that will be used by default.

SimpleGlobalAccountMapper

This global account mapper implementation expects the account data in a schema that matches the *SimpleClientAccountMapper* on the server:

Example account data:

```
accountIdentifier: 'jdoe'
roles: ['Vendor.MyPackage:User']
party:
  __type: 'Vendor\MyPackage\ExampleParty'
  company: 'Acme Inc.'
```

The account mapper will instantiate a new *Account* instance that is transient and *should not be persisted* as for every single sign-on process a new Account will be created. If the authenticated account needs to be referenced by other domain models a custom global account mapper implementation has to be created that could create new persisted accounts lazily and update their data according to the given account data. Because this is very domain specific we do not ship a default implementation for this mapping strategy.

The party type is given as the key *party.__type* and will be mapped to a type on the instance using the *Flowpack.SingleSignOn.Client.accountMapper.typeMapping* setting. This setting allows to have a simple one-to-one mapping between classes on the server and on the instance. The property names in the account data have to match for the default implementation.

1.4.7 Logging

The client package performs no special logging besides a logging aspect for signed request debugging. The aspect to log failed signed requests with signature verification errors can be enabled via the *Flowpack.SingleSignOn.Client.log.logFailedSignedRequests* setting. The requests are logged to the Flow security logger.

1.4.8 HTTP services

This is a list of all HTTP services (controller actions) that are exposed by the client. The URI path depends on the global Routes.yaml that mounts the package subroutes, we expect the routes to be mounted at */sso/<SingleSignOnSubroutes>*.

Public

/sso/authentication/callback Route for the *authentication callback*, has to be accessible for all users that should authenticate using the single sign-on.

Private

The controller for these routes are protected by a signed request firewall filter and should only be accessible by a single sign-on server. We strongly suggest to take additional measures for securing the server-side channel between the server and instances (e.g. SSL with client certificates, firewall rules, additional request filter).

> **Warning:** The default TYPO3 Flow routes could allow access to controller actions even though the URI paths are secured by a firewall or webserver configuration.

/sso/session/{sessionId}/destroy Route for the *Client notification* to destroy the local session on an instance when a user logs out on another instance or the server.

1.5 Development

Development of the Flowpack.SingleSignOn packages and distributions is coordinated on GitHub: http://github.com/Flowpack

1.5.1 Running the tests

We have a test suite that covers all scenarios of the single sign-on with acceptance tests through Behat in the TestSuite repository. The tests need a running demo setup with two different instances (configured via subcontexts).

Install Behat via Composer:

```
$ git clone https://github.com/Flowpack/Flowpack.SingleSignon.TestSuite.git TestSuite
$ cd TestSuite
$ path/to/composer.phar install
```

The default *behat.yml.dist* configuration expects the demo installation with the URL *http://ssodemoinstance.dev/*, *http://ssodemoinstance2.dev/* and *http://ssodemoserver.dev/*. A custom configuration for Behat can be used by copying the file *behat.yml.dist* to *behat.yml*.

Running the Behat tests:

```
$ bin/behat
```

This should execute all features and display the results of the scenarios.

A

Account
- Impersonation, 20

C

Client, 21
- Account mapping, 26
- Authentication callback, 26
- Authentication provider, 22
- Commands, 24
- Configuration, 22
- Entry point, 22
- HTTP services, 27
- Installation, 22
- Key pair, 22
- Logging, 27
- Registration, 16
- Servers, 22
- Service base URI, 22
- SimpleGlobalAccountMapper, 27

Command
- ssokey:exportpublickey, 25
- ssokey:generatekeypair, 24
- ssoserver:registerclient, 15
- ssoserver:removeexpiredaccesstokens, 15

D

Demo, 6
- Credentials, 6
- Instance, 10
- Server, 8
- Setup, 6
- Vagrant, 6
- Walkthrough, 8

S

Server, 12
- Access token, 17
- Access token redemption, 18
- Account mapping, 18
- Authentication callback, 17
- Authentication endpoint, 16
- Client notification, 20
- Client registration, 16
- Clients, 13
- Commands, 15
- Configuration, 13
- HTTP services, 21
- Key pair, 13
- Logging, 15
- Logs, 15
- Redeem access token, 18
- Service base URI, 13
- SimpleClientAccountMapper, 19

Session
- global, 20
- local, 20
- Synchronization, 20

Single sign-off, 20

V

Vagrant
- Demo, 6