Dataverse - SPA Authentication Requirements and Design Options

Introduction

Authentication is the process by which a user proves their identity to an application, e.g. with the expectation that they will then be authorized to act on protected resources. In general, authentication is usually broken into two steps for efficiency: an initial login involving a username/password, two-factor authentication, or other mechanism in which the user's credentials are initially validated, and a lighter-weight mechanism to revalidate the user in subsequent requests to the application. The latter usually includes a timeout mechanism after which login must be done again (or, optionally, a separate refresh mechanism may be invoked).

In Dataverse today, initial login is only supported through the user interface (UI) with subsequent revalidation relying on a standard session cookie. Dataverse supports login via several mechanisms, discussed in a later section, that allow Dataverse to be run as a stand-alone application or leveraging institutional, community, or global authentication mechanisms.

Direct application programming interface (API) access requires an initial UI login to obtain one of several types of credentials, described in later sections, that are geared towards different use cases. These are transferred by Dataverse-specific mechanisms and provide access to some or all API calls for various durations. (The one exception to this is that the API call to create a new built-in user returns an APIToken. This is used, for example, in API tests, but having an admin create an account and send the APItoken to a user would be a way for them to never have to use the UI.)

The development of a single page application (SPA) front-end for Dataverse, and, optionally, supporting the ability to use parts of an SPA and the current UI together, involve requirements that cannot easily be supported via Dataverse's current authentication mechanisms. This document outlines these new requirements and design options to meet them.

Requirements

When evaluating different solutions for supporting SPA user authentication, the following requirements and desirable features have been taken into account:

- **API-First**:
    - To support an SPA, or other UI, and to support API access independent of any UI, it must be possible to login independent of the current Dataverse UI while continuing to support authentication to the Dataverse back-end (API service).
    - Designs should use standard authentication mechanisms and minimize the Dataverse-specific elements.

- **Backwards compatibility**:

    - Support all authentication providers currently in use in the community
    - Support current API authentication mechanisms, such as API and workflow keys, private URLs, and signed URLs.
    - JSF Frontend pages should be accessible during the gradual SPA migration, and the new SPA frontend should be interoperable with the JSF frontend pages.

Current Authentication Providers

To better understand the architectural approaches described in this document, it is necessary to take into account the different login protocols and authentication providers currently supported by Dataverse. These are the following:

- Internal Users (Built-In users)
- OIDC Users (e.g. institutional logins and federations)
- SAML / Shibboleth Users (e.g. institutional logins and federations such as InCommon)
- OAuth Users (ORCID, GitHub, Google, and Microsoft)

All of these mechanisms are in use across the Dataverse community. For internal users, Dataverse directly stores the username and (hashed) passwords. For the other mechanisms,

Dataverse uses the indicated protocols to validate a user and tracks the persistent user id from the provider and source authentication provider in its database.

As noted, the API does not currently support any of the authentication mechanisms described above, as their use is tied to the JSF Frontend, which supports them as different options for the Dataverse Login Page.

While direct support for the specific protocols listed above is not required, a new design should retain the ability for the underlying authentication providers to be used. Further, to support development, testing, and stand-alone deployment, it is important to minimize the need for additional components to support built-in users.

Current Revalidation Mechanisms

Before describing re-architecture proposals, it is important to understand how the current Dataverse authentication mechanisms work and therefore where we start the re-architecture work from. All of these mechanisms involve a secret generated by the Dataverse back-end that, when returned in an API call, proves the user's identity and allows subsequent authorization decisions.

Dataverse currently supports 5 revalidation mechanisms, two used by the UI and 4 applicable for API calls:

1. Session Cookies: The Dataverse UI generates a session cookie for all users (before login, users have a cookie identifying them as a 'guest' user) that is returned in subsequent requests for new pages or partial page updates (via AJAX requests). Session cookies have a configurable, relatively long lifetime, e.g. 8 hours by default (Link to docs).
2. API Keys: Dataverse can generate a random token for a user, which can be read in the UI, that can be sent as a URL parameter or in a custom HTTP header in API calls. Dataverse is able to do a database lookup of the API token to identify the user. API Keys have a very long lifetime (1 year).
3. Workflow Tokens: When Dataverse runs a workflow on behalf of a user, it can send a generated random workflow token to any external application involved in that workflow. The workflow token can be sent with an API call via mechanisms analogous to those for an API Key - as a URL parameter or custom header. Workflow Tokens differ from API Keys primarily in their lifetime - they are only valid until the workflow completes. (There is also no way for a user to obtain a workflow token directly.) As with API keys, Dataverse does a database lookup to associate a user with the workflow token sent.
4. Private URLs: Private URLs include a random token that allows a guest user to view a draft dataset and to view and download files within that draft dataset. (A variant - anonymized private URLs - helps support anonymous review by suppressing metadata

such as author and contact information.) Users can create a Private URL for one of their draft datasets via the UI. The token for a Private URL is passed as a URL parameter or as a header, like API tokens. Dataverse does a database lookup to identify the user and dataset involved.

5. Signed URLs: Signed URLs are stateless constructs that provide access to a specific API call, as a specific user, for a specified amount of time. The primary mechanism for signedURL creation is via Dataverse's external tool mechanism. Dataverse generates specific URLs when a user launches a tool via the UI and sends them to the specific tool for use in callbacks to Dataverse. Signed URLs can also be generated via an API call. Signed URLs provide cryptographic proof of the user's identity and do not require a database lookup. Nominally, like Private URLs, Signed URLs mix authentication and authorization in that they limit access to a subset of what a user is allowed to do (in this case to making the one specific API call the Signed URL is designed for).

Dataverse API authentication is currently highly centralized within *AbstractApiBean.findUserOrDie* method, where an authenticated user is searched for each of the different supported credentials: API Token, Workflow Token, Private URL, or Signed URL. If no credentials are provided, a guest user is returned. The one exception to this is that the file download API call can also be used with a session cookie thanks to custom code for that specific call.

AbstractApiBean is an abstract class from which the different "API controller" classes for different Native API endpoints inherit, and the *findUserOrDie* (and similar) method(s) is called from any endpoint method that requires user authentication (Example).

New API Authentication Mechanisms

Design Considerations

The challenges in supporting authentication with an SPA stem from two facts:

- the SPA itself is separate from the back-end Dataverse instance and login must allow the user to seamlessly authenticate to both the SPA and the Dataverse back-end, and
- because the SPA runs in the user's browser rather than on a trusted server, authentication mechanisms that require the SPA to store application-level secrets (versus user/session secrets) can be attacked. (Application-level secrets are those the application must provide to the authorization service (either directly or by signing a message) to be allowed access. If they are compromised, the application can be impersonated.)

Another consideration with an SPA involves the revalidation mechanism. Cookie-based mechanisms, such as the session cookie used in Dataverse today, can be vulnerable to Cross-site request forgery (CSRF) attacks when supported for public API calls. While this can be mitigated, it adds complexity to cookie-based mechanisms. Conversely, mechanisms that require database lookups, as many of Dataverse's current mechanisms do, can cause slow response in an SPA where many API calls may be needed to construct the display. Again, while this can be mitigated, it adds complexity.

Together with the requirements and desired features listed above, these concerns constrain the potential designs for an SPA authentication mechanism.

Login Options

Given the requirements and concerns, there are two primary options for providing the currently supported suite of authentication providers with an SPA. The first would be to essentially maintain the existing Dataverse UI login capabilities, or replicate them in a new component, and send the SPA some form of revalidation token for use in subsequent API calls. This is not very desirable for several reasons:

- it requires maintaining a separate technology stack for the login pages, making the maintenance of a common look-and-feel more difficult
- it requires maintaining support for all the supported authentication protocols in the Dataverse codebase (this has been done to date, but other design options would simplify Dataverse code), and
- it doesn't provide a way to easily support additional apps or login via API.

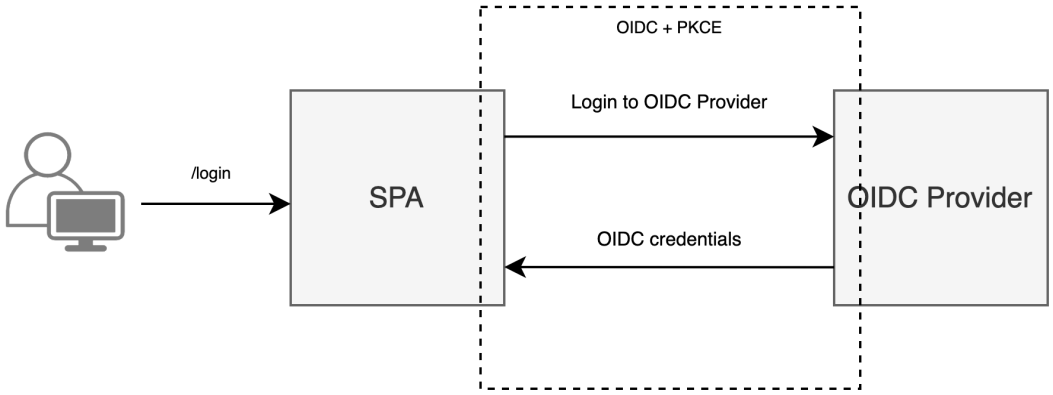
Given these drawbacks, the second option - using a protocol specifically designed for SPAs and leveraging an external component to provide support for other protocols - is the one we've investigated more thoroughly. The core design for this approach and its variants are described next.

OpenID Connect (OIDC)

OIDC builds on the widely used OAuth 2.0 authorization protocol, adding support for authentication and standardizing the format in which applications receive user information such as name and email. OIDC is already supported as a UI login mechanism in Dataverse. Further, when used with Proof Key for Code Exchange (PKCE - RFC 7636), OIDC is able to address the security concerns related to an SPA running in the user's (untrusted) browser.

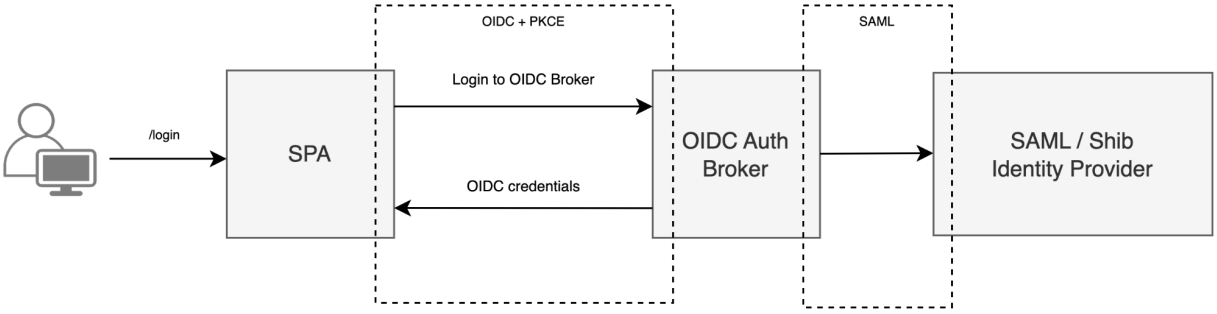
The basic flow when using OIDC is shown below. When a user logs in to the SPA, they are redirected to the OIDC provider website where they enter their credentials (username/password,

multi factor authentication, etc. as required by the provider) and proof of who the user is is returned. When an SPA is used with a backend service such as Dataverse, there are multiple options for exactly how to implement this flow.

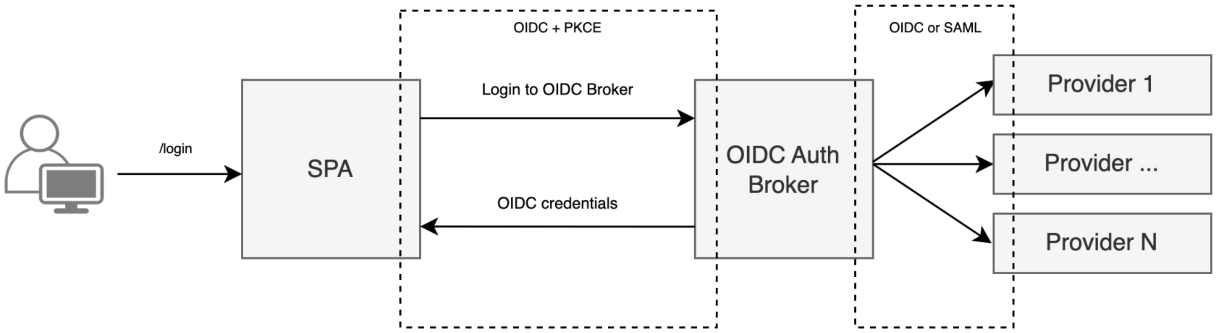


Before turning to those details, it is useful to note that OIDC also provides a way to support Dataverse's other existing login mechanisms. OIDC "Brokers" exist that can connect to authentication providers using other protocols, i.e. SAML/Shibboleth, OAuth, and, with some additions, Dataverse's built-in user authentication. While it may still be worthwhile to provide a way to run Dataverse using built-in user accounts without requiring an OIDC Broker, OIDC should be a useful way to outsource support for the other protocols:

- multiple open source broker options exists
- nominally configuring an external broker would be about the same complexity as configuring them directly in Dataverse (with more possibilities for support from the broker developers)
- given OIDC's popularity, brokers may be run at the institutional level, meaning Dataverse administrators would not have to manage a broker directly.



Since OIDC brokers can also federate multiple providers, Dataverse would potentially no longer need to support a list of providers internally or to track the provider associated with a given account, which would result in further simplification.



Option 1: OIDC PKCE in the SPA

In the normal PKCE flow, the OIDC Provider or Broker returns at least two tokens to the SPA, an ID token and an access token, with an optional refresh token. The ID token is a signed statement providing the user's name, email, and related information. It is proof that the user has successfully logged in to the specified application (the "aud - audience" included in the token). According to the specification, the ID token should not be passed to another application. (However, some implementers, such as Kubernetes, decide that their SPA and backend are one application and do send the ID token to the backend.) Instead, the recommended practice is to send the access token. However, by default the access token does not have information about the user (i.e. name, email) that Dataverse would require to create an account/associate the user with an existing account. There are at least two options to manage this:

1) Depending on the provider, OIDC allows access tokens to include name/email information -or-
2) OIDC defines a /userinfo endpoint. With the access token, the Dataverse back-end could call this endpoint to retrieve the user's name/email.

(Where 2) is covered by the standard, 1) is a provider specific feature)

Option 2 OIDC PKCE involving the Dataverse backend

One of the limitations in the first option, which is often accepted in stand-alone SPA implementations, is that the tokens (ID, access, and possibly refresh) are stored in the user's browser where they are potentially vulnerable. An optional also exists with PKCE for the SPA to initiate the process and to then pass information required for the process to complete to the back-end (one example). The back-end is then the direct recipient of the tokens and it can use an alternate mechanism to allow the SPA to revalidate to the back-end after login. (Options for revalidation are discussed later). In this case, the back-end can use the ID token directly to find the user information required to create an account or associate the user with an existing account.

Built-in user Login

As noted above, it should be possible to allow login to built-in accounts (where Dataverse stores the username/hashed password) via an OIDC broker. For this to work, Dataverse would need an API endpoint for built-in login, i.e. where the OIDC broker could send a username/password via HTTP Basic Authentication and get a response from Dataverse indicating success. (see the Account Creation section for notes about registering new users.)

Nominally, the same endpoint could be used by the SPA directly to login a built-in user. For this, the SPA would have to implement a username/password login interface separate from the OIDC option. This could allow installations (dev, test, or production) using only built-in accounts to avoid having to configure an OIDC Broker. It may or may not be worth allowing this option when OIDC is also in use. In any of these cases, once login has succeeded, it should be possible to use the same revalidation mechanism as with OIDC logins.

Making the Dataverse back-end an OIDC provider was considered but appears to involve significantly more work than the alternatives above.

Account Recognition/Backward Compatibility

In Dataverse today, user accounts must have a unique email and are only allowed to be associated with one authentication provider (i.e. a user with a given email cannot login both through Google via OAuth and via an institutional Shibboleth provider). With an OIDC broker, or in cases where a Shibboleth provider is replaced by an OIDC provider, both options above would allow the Dataverse backend to discover the user's email and thus associate it with an existing account. However, it is unclear if/how information about the underlying provider will be conveyed and how Dataverse would map this information to the existing provider information (i.e. as stored in the authenticateduserlookup table).

One option would be to stop tracking the provider. This would nominally change the behavior of Dataverse, i.e. allowing someone who had logged in via a multifactor institutional provider to log in again via a password-only Google or Github provider (assuming these were all supported). (OIDC login for current built-in user accounts could also be possible).

An alternative would be to assure that provider information is provided via OIDC and to then map existing provider information (either via bulk update or upon login) to replace legacy values in the Dataverse database. This option would preserve the existing behavior tying accounts to specific providers.

Account Creation

In addition to allowing login by existing users, our chosen design must support creation of new accounts. For OIDC, account creation could potentially be automated: if the unique identifier (“sub”) or as a potential fallback the email returned via OIDC is not yet in use, a new account could be created using the information in the ID token or returned via the /userinfo endpoint (depending on the OIDC implementation option chosen). It would also be possible to retain the distinction between registration and login (login would fail if no account exists and registration would fail if one does and the user would be directed to register/login respectively). All of this assumes that the OIDC provider is configured to return all of the user information required for Dataverse to create an account, or, when the provider is a bridge to other realms like a SAML federation, asks for missing required information.

For built-in accounts, it is not yet clear if OIDC brokers can support account creation directly. If not, an SPA interface to create an account would be needed. This would also be needed to provide built-in user support without having an OIDC broker configured. In any case, Dataverse would need a new API endpoint to support builtin account creation (An API endpoint for this exists now but it is intended for use by administrators, requires configuration of a special key that would not be safe if shared with the SPA. In contrast, what is needed is an endpoint that requires an authenticated user to make the request.)

Revalidation

With either of the OIDC options identified above, and with direct built-in login, it would be inefficient to replay the initial back-end authentication process for each subsequent API call from the SPA (or other client). Instead, some form of session or stateless token would avoid the database lookups and/or callbacks required in the first call.

Session-based validation:

Dataverse uses session-based authentication to support its current UI. This is implemented via Java's standard HttpSession mechanism which uses an opaque cookie with a set lifetime (usually hours) sent with every request. Internally, the opaque cookie is associated with cached information, i.e. the user's identity. Using this mechanism with API calls opens up additional security concerns (related to Cross-site Request Forgery (CSRF)) that can be mitigated by the use of anti-CSRF tokens and/or use of the SameSite attribute on the cookie.

Stateless Json Web Tokens (JWT):

One of the drawbacks of cookies is that the server (Dataverse back-end) needs to cache the association between the cookie id and the user's information, which requires memory allocation per user that limits scalability. Stateless tokens, such as JWTs avoid this by including the user information in the token itself, relying on cryptographic signing and validation to assure the token is valid. This increases the amount of information transferred in each API call, and requires processing to validate the signature, but eliminates the need for server-side storage outside the duration of a given API call. JWT is a standard format for such tokens that is supported by multiple libraries, including ones already in use in Dataverse. JWTs are still vulnerable to being stolen from the SPA, but they are more secure than vanilla cookies due to browser behavior: by default cookies are sent to the server from the browser regardless of whether the page making the request is part of the SPA or some other site the user is browsing, whereas JWTs sent as HTTP Authorization:Bearer tokens, would not be sent by the browser with other non-SPA requests. (The SameSite attribute for cookies changes the default to not send the cookie with requests from other sites.)

These two options would work regardless of whether OIDC or local login of built-in users is used for initial authentication. It would also be possible in the OIDC case where the SPA holds the access token only to continue to send the access token on subsequent requests. This would require the Dataverse back-end to either cache the association of access token and user account (as with a cookie), or repeat calls to the OIDC /userinfo endpoint for each request. As this mechanism is not general and doesn't provide much of an advantage over the others (though see the Refresh section below), it is probably not a viable choice.

Access Token as a Stateless Token

Nominally, access tokens are intended to be opaque. However it is possible for them to be JWTs with additional information. If the access token is treated as opaque in dataverse, revalidation would require

- calling the OIDC /userinfo endpoint on every call, or
- Caching the access-token to user identity mapping (with scaling concerns as with sessions), or
- Using it in concert with some other session/token, or
- Requiring it to be a JWT with the required user information in it.

The last option would be easiest to enforce if we always assumed that an appropriately configured KeyCloak instance was available (i.e. to create a JWT access token if the underlying OIDC provider does not provide one). The other options would probably be more work and/or less efficient than sessions or JWTs. However, all of these would retain the OIDC pattern of using an access token on each call.

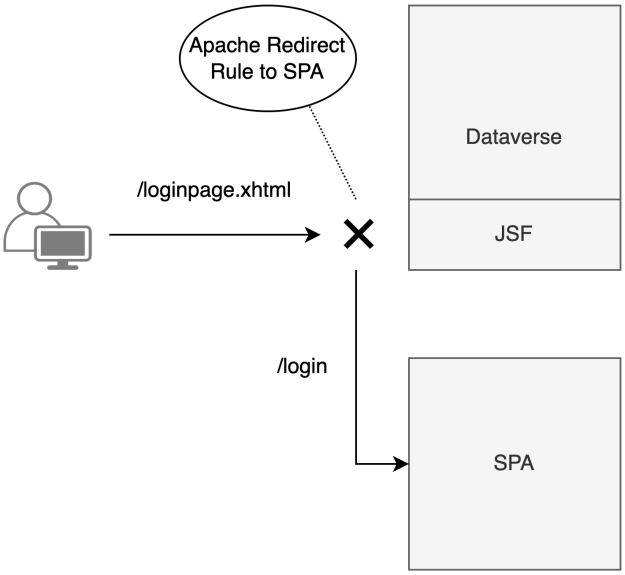
Interoperability of the SPA and Current Dataverse UI

Session-based revalidation as described above would be compatible with the current Dataverse UI. For example, if a user used the SPA to login and view a Dataverse, the SPA could redirect the user to the current Dataverse UI's Dataset or DataFile pages and the session cookie would allow access to the protected draft versions. If the SameSite attribute is used, the SPA and Dataverse API would have to be served from the same domain (two related subdomains could be made to work if the cookies are set on the common parent domain.) While this type of interoperability is not a requirement, it is a potentially desirable feature that could support incremental adoption of the SPA, simplify side-by-side comparison of SPA and current UI pages, etc. (Although it may not be needed if authentication work is prioritized, this could also allow other SPA work to proceed - users could login through the current UI and the SPA would share the session such that the user would be logged in there as well.)

In the second scenario, using JWTs, this would not be possible without additional work. The current UI and SPA could run from the same back-end Dataverse instance, but if the UI sets up a session and the SPA uses JWTs, logging into one would not automatically log the user into the other. There would be several options to enable such interoperability:

- create a session in addition to returning a JWT upon SPA login
- create an API endpoint to retrieve a JWT given a session to which the SPA would make an initial call
- support sessions and JWT across the API

These differ in whether Dataverse UI or SPA login is required and, in combination, could support login from either starting point. If one starting login point were selected, the other could be forced to redirect to it. For example, to only allow login via the SPA, traffic to the Dataverse /loginpage.xhtml page could be redirected to the SPA login, e.g. via an Apache redirect rule.



Refresh/Revocation

Currently, Dataverse relies on sessions that have a configurable but usually fairly long lifetimes (e.g. hours to a day). With an SPA/backend design, where sessions and/or tokens are potentially more at risk, shorter durations could be a useful practice. OIDC can be configured with access tokens only living a few minutes. In this case, a refresh token is also sent. It can be used to contact the OIDC provider for a new access token without requiring the user to login again. In the event that a user's account with the OIDC provider has been disabled, the refresh request would fail, meaning access to the SPA and Dataverse would also cease within minutes instead of hours.

It is not clear that our design needs to address this. If it is desired, the specific mechanism to implement it would depend on which OIDC option we choose and whether direct login for built in users is supported. Nominally, any design would need to involve using short sessions or JWT tokens (or using the access token directly) and a mechanism for the holder of the access token (the SPA or Dataverse back-end) to request a refresh. (If the Dataverse back-end receives the OIDC tokens directly, it would have to store the refresh token for a user during its lifetime.) Similarly, for local built-in user login, some refresh mechanism would need to be defined, although a design could only support refresh when built-in login is done through OIDC.

Logout

Currently, Dataverse handles logout by destroying the user's session when a logout request is received. With an SPA or other client, the logout mechanism would depend on the design. If revalidation is done via a session, adding a Dataverse API call to destroy that session could suffice. Conversely, if stateless tokens are used, having the client destroy the token could be sufficient. In either case, if the initial OIDC login uses prompt:none (which allows passive login - login without prompting the user if they have recently logged in at the OIDC provider), then use of the OIDC logout mechanisms may be required.

Refactoring in Dataverse

As explained, for UI interactions Dataverse uses a session managed by the servlet framework but Dataverse API authentication is done in the *AbstractApiBean*. Rearchitecting to make the API accessible from an SPA provides an opportunity to also refactor this code to increase modularity. Specifically, instead of calling the *findUserOrDie* method on each secured endpoint to authenticate the request through one of the possible authentication credentials, this responsibility can be delegated to an outer layer. Since Dataverse already uses the JAX-RS

standard in its API, the layer would nominally be a JAX-RS *ContainerRequestFilter* doing authentication/revalidation as a *@PreMatch* operation which would either reject the request (if authentication fails) or pass the request to the underlying API method with the logged in user account info included. To mark API endpoints that require authentication and therefore filtering, a custom annotation can be used.

It may also be useful to move support for Dataverse's existing API authentication methods into such a filter (or another one(s) - filters can be stacked). This may be most straight-forward for the stateless SignedURL mechanism. The other mechanisms, which require Database lookups, could also be moved to a filter but would still have dependencies on internal Dataverse code.

Prototyping Efforts

In creating initial proof-of-concept (PoC) implementations of the SPA and to inform the creation of this document and to start identifying specific technologies to use, PoC implementations of components relevant to authentication have been done.

Session-based Revalidation

Given that Dataverse already supports session-based revalidation for the UI, a simple way to make the API accessible is to also support sessions there. As noted, this would force the API to manage a server-side stateful session, which limits scalability and breaks the statelessness intended in REST designs.

Making the API accessible via a session cookie and having to manage a cookie in the SPA presents CSRF risks that would be mitigated in other API authentication mechanisms such as Bearer tokens sent in request headers. In the initial PoC, a filter-based protection mechanism was developed on top of the API to prevent these attacks, which would force the SPA to have to manage an anti CSRF token. SameSite cookie annotation, which could be used in addition to or as an alternative to a CSRF filter, was not tried.

The PoC also added an endpoint to allow login to built-in user accounts. This allows login to those accounts via the SPA whereas other types of login are only supported via the existing Dataverse UI. (This is a limit of the PoC rather than of the design using sessions.)

OIDC-PKCE

A PoC was created for the OIDC-PKCE login option in which the SPA handles the entire process. Initially, the PoC used react-oauth2-pkce, which was tested on the React PoC SPA. After experiencing issues with react-oauth2-pkce in the PoC when handling different

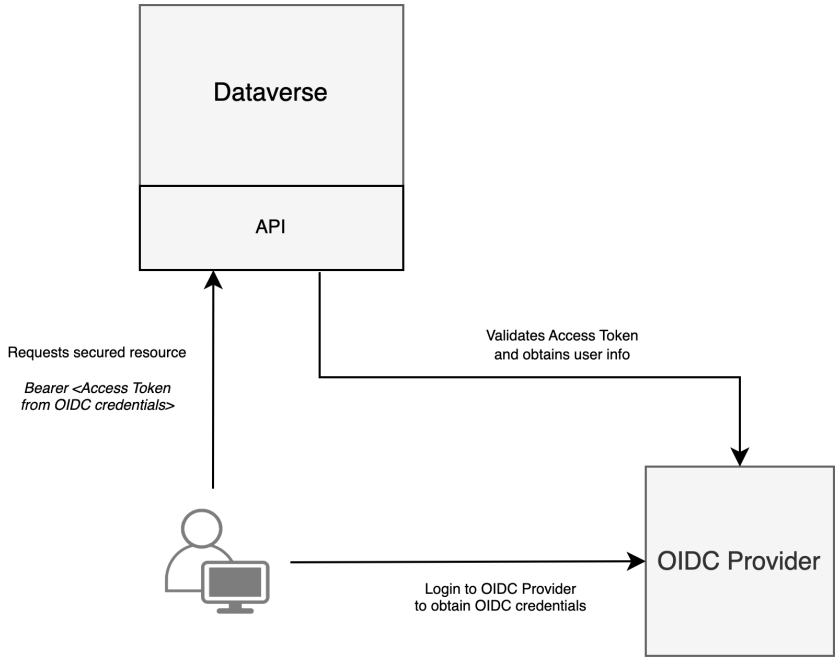
authentication states (these problems were also experienced by members of the Dataverse community (Johannes & Vera who opened a GitHub issue), the PoC was switched to use AXA's react-oidc. We consider AXA's react-oidc as the best option for the moment. It has already been tested by Johannes & Vera and is built upon the official openid/appauth package.

There is an old issue about PKCE support for OIDC / OAuth2: https://github.com/IQSS/dataverse/issues/8349.

OIDC Auth Broker

There are multiple OIDC Authentication Brokers available including the open source KeyCloak and offerings from Okta/Auth0 that are free for smaller installations. A PoC was developed using KeyCloak, making it deployable as a docker container. This provides an easy installation for, for example, a local development environment. The SPA code in the PoC is abstracted from the specific implementation of the OIDC provider and should also work with institutional OIDC providers, etc.

In the PoC, once we have obtained the credentials of an authenticated user through PKCE-OIDC in the SPA, it is necessary for the SPA that the Dataverse API has the capability to authenticate OIDC users. An initial PoC sending the access token to the Dataverse backend is in development. As this capability (to support OIDC access to the API) has been requested independent of the SPA work, the changes to Dataverse are currently being developed in a PR (#9230). This PR includes using the access token to perform a callback to the OIDC provider to obtain user information to allow matching to an existing account. The PR implements checking of the OIDC access token as part of the processing in the AbstractApiBean using the Nimbus library which is already used elsewhere in Dataverse. As the work is in progress, the performance impact of performing a callback for each API call has not been assessed. As noted above, although the SPA also has the identity token, sending the access token is conformant with the OIDC specification.



Keycloak OIDC Docker configuration

To simplify development and testing of OIDC support, work has also been done to provide a default Docker installation of KeyCloak. Since this is also useful for testing Dataverse's existing OIDC login capabilities, it has been submitted for inclusion in the Dataverse codebase as PR #9234. The docker container can be started via a script. New users can be added via the Keycloak UI for testing.

Auth Filter

The new filter-based authentication design adds a filter on top of the Dataverse API, which is intended to verify that, for those API endpoints that require user authentication, a valid credential is provided for any of the supported API authentication mechanisms.

The implementation is currently in the develop branch, in a new api/auth folder. In particular, we can find the following components in that folder:

- *AuthFilter*: Filters requests to secure endpoints by verifying that a valid authentication credential has been provided. It is intended to act in a similar way to *findUserOrDie*, by doing a cascade filtering of the different supported credential types.

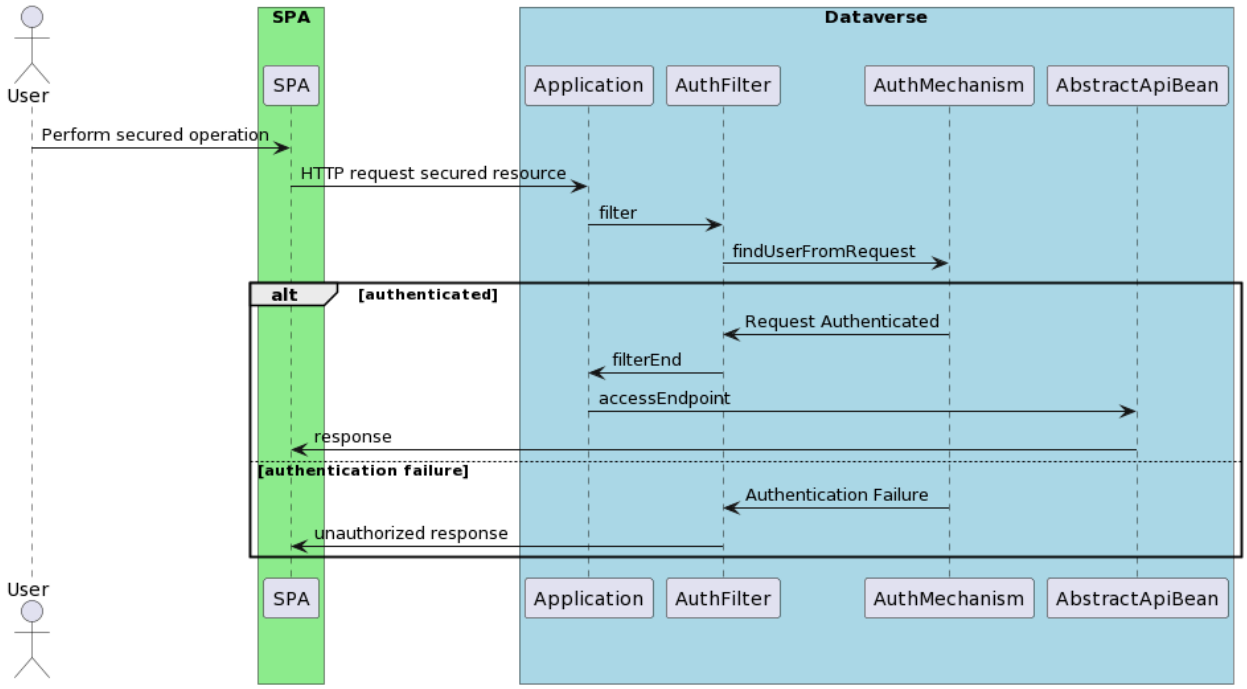
- *AuthRequired annotation*: New annotation shared between the filter and any secured endpoint method.

- *Auth mechanisms*: There will be as many as types of API credentials to validate.

This design extracts authentication logic from the *AbstractApiBean* to a higher level and encapsulates it into different components with unique responsibilities. This improves the testability and extensibility of the authentication mechanisms.

Extensibility is a very important requirement that has been taken into account for this solution, considering a future auth mechanism for OIDC, which would validate an OIDC access token and obtain the associated user. There is an open PR for this mechanism: https://github.com/IQSS/dataverse/pull/9532

Below is a sequence diagram where we can see the components described above in action.



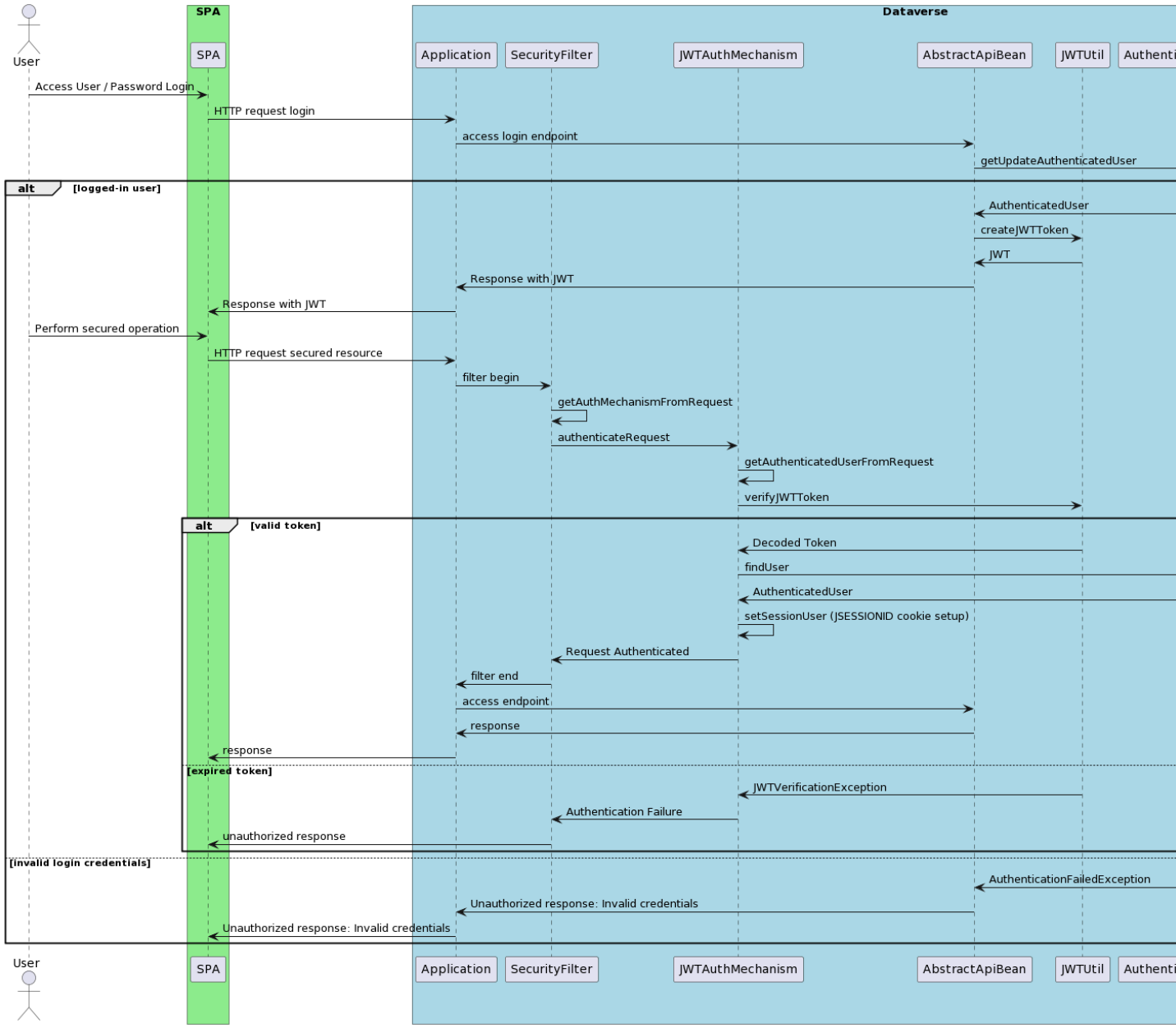
Auth for Built-in users

There is a PoC based on a primitive version of the previous design that includes the ability to authenticate to the API with built-in accounts via an HTTP Basic Auth mechanism: Link to repository (This is also supported today in the Dataverse SWORD API.) As noted, authenticating on each API call is less efficient than using a revalidation mechanism. In the context of an SPA, it is also not a best practice since it requires the SPA to store the long-lived, unencrypted form of the username/password in the user's browser. Further, the use of HTTP Basic Authentication itself is problematic as browsers will cache the credentials and resend them on subsequent calls to the same API, making logout complicated.

A second PoC, building on the first, has been created that demonstrates the use of JWT tokens as a revalidation mechanism. It includes an API call to allow login (taking JSON username/password values rather than using HTTP Basic) that returns a JWT with a 15 minute lifetime that can be sent with subsequent API calls as an HTTP Bearer token to revalidate. Validation of the JWT is done via the security filter. This approach is analogous to the OAuth Resource Owner Password Credentials flow. In the PoC, the JWT includes the database id for the user account. If the signature is valid and the signature uses Dataverse's private signing key, and the token has not expired, Dataverse can use the id to associate the request with the correct user. Other user information, e.g. name and email, could be added to the JWT to avoid the SPA having to make additional API calls.

The PoC uses auth0/java-jwt, which is a Java implementation of JSON Web Token (JWT). This library is already part of the current Dataverse dependencies (used only to support archiving to Harvard's DRS which may be removed from the core codebase at some point). JWT support is also available in the Nimbus library which may be a better long-term option for use in the security filter.

Below is a complete sequence diagram of this flow, including both user login and secure endpoint access:



Complete Design Options

Based on the information above, the entries in this section are intended to provide complete design options that address all of the areas of functionality and concerns outlined above.

Option 1: OIDC PKCE in the SPA with JWTs, direct support for builtin users, and backward compatibility with the current UI

Components

- OIDC PKCE performed in the SPA
- Dataverse API call to login to the API, i.e. exchange OIDC access token for long-lived JWT token
- Subsequent calls include JWT which is validated via a filter. (Based on the Auth Filter design)
- Built-in users supported via OIDC or, for simple installations, via a second login API call that exchanges username/password for the same long-lived JWT token
- Logout would involve dropping the JWT token (no logout API call because there's no server state)
- Refresh would not be supported
- Current UI would be supported by also including a session cookie in the response from the login calls (same duration as the JWT)
- Built-in user account creation would be handled by SPA via a new API call. (Minor option - current UI registration form could be used to start and avoid the extra SPA work for now/until the larger question of whether built-in users should continue to be supported is resolved)

Pros

- Consistent with current prototyping
- Less work than (some?) other alternatives
- Dataverse becomes stateless w.r.t. User sessions once current UI is retired, increasing horizontal scalability
- Avoids more complex logout/refresh functionality, but with shorter JWT/session cookies could be consistent with adding refresh/logout later

Cons

- The SPA/other clients have to understand Dataverse's JWT mechanism (though they do just treat them as opaque bearer tokens)
- Revocation of the JWT is not easily possible (could change signing key), so other (untrusted) clients would have ~8 hour access

Variant: OIDC PKCE completed in Dataverse

Components

As in Option 1, except:

- PKCE is completed on the backend, so Dataverse gets the ID and access token directly

- SPA would get the JWT/session cookie directly at the end of the PKCE process and would not have the accessToken
- Dataverse would drop the accessToken after login and would not attempt to refresh

Pros

- ~more secure (no accessToken/refresh token in the browser)
- Avoids Dataverse having to make a second call to the /userinfo OIDC endpoint

Cons

- More complex
- Security gain is less if JWTs are long-lived (could make JWTs short-lived and have Dataverse do refresh with the accessToken)
- May not be supported by available OIDC libraries
- More complex for other clients

Variant: OIDC PKCE using only the accessToken

TBD - as in Option 1 but the SPA always uses the short-lived access token (no JWT, possibly still using session cookie temporarily for current UI compatibility) - would allow revocation, and would require refreshing and logout.

Option 2: Keycloak as required element + OIDC PKCE completed in Dataverse

Components

- Keycloak as required architectural element
- Migrate internal users to Keycloak from the beginning: Being username/password users they follow a structure similar to Keycloak internal users. Keycloak API can ease this migration process, by exposing an API with user management endpoints such as user creation.
- Preserved user behavior for new installations: by including the dataverseAdmin user in the base configuration specified in the Keycloak Realm file or any other required element.
- Centralized login into the SPA (and disabling it in JSF)

Pros

- By using Keycloak as a unique OIDC provider, we ensure that we will always receive a JWT-formatted access token for revalidation. It doesn't matter what the source of the user is (external OIDC provider, external SAML provider, user/password), since Keycloak is an OIDC provider that we control.

- By being all SPA users OIDC users and in combination with the PKCE flow involving the backend for more security (Option 2), we can achieve a solid and secure authentication flow.
- We can greatly simplify the Keycloak installation process thanks to Realm base configurations

- It also makes the architecture more modular and secure, removing any form of user authentication from Dataverse Backend and leaving it just as a Resource Server. (This is true of option 1 as well)
- In cases where there is no central OIDC-enabled provider, people would need to provide a broker anyway. It's not clear how many installations out there will have the ability to re-use one of these, so maybe this isn't such a large step.
- There might be a possibility to use this intermediate component to ensure certain attributes like email addresses etc are present (at least Unity IDM enables this, Keycloak unknown), which removes this burden from the application and the SPA themselves.
- Even if an upstream OIDC provider would not support multi-account auth to the same user, having this intermediate component in all use cases might enable this (also this is hard to do right - even IDM people stepped back from this, i.e. for Helmholtz AAI!).

Cons

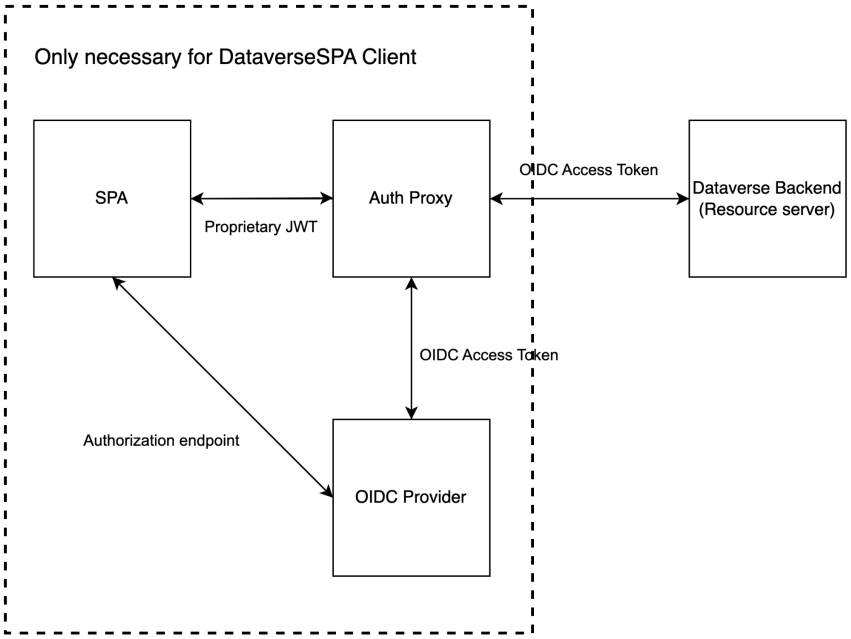
- This approach would force installations to include Keycloak as a required architecture piece, even if they don't need to manage external OIDC users or just want to access one external OIDC provider.
- It might inflict a larger one-time migration effort for existing installations
- It limit's our OIDC compliance in that we require a specific form of OIDC token
- It would require dealing with refresh tokens up front unless we can also require Keycloak to provide long-lived access tokens

Variant: same as above but with PKCE centralized in the SPA (as in Option 1)

Variant for both Option 1 and Option 2: OIDC PKCE completed in Dataverse with third component

For the part of the backend that performs the PKCE flow, this approach is intended to encapsulate this responsibility in a separate architectural component.

For the implementation of this service, a pattern such as Backend for Frontend Proxy or Token Mediating Backend can be applied but there may be other patterns that fit and we would need to analyze.



This Diagram is a simplification of the "Backend For Frontend Pattern" applied to Dataverse

Since Dataverse works as a Resource Server that expects an access token, the "public client" option is supported for those clients that want to manage PKCE entirely, for example a custom SPA other than Dataverse SPA.

With this approach Auth Proxy is still available for clients who want to have a confidential client, but it is not mandatory.

Pros

- Limited responsibilities: Dataverse operates just as a resource server, abstracting from that authentication part, which is delegated to a separate component.
- Modularity: Considering we provide this "third component" implementing OIDC-PKCE, other installations that prefer to implement another authentication flow can do so by replacing this component.

Cons

- Another component to manage
- ~minor - Communications on either the first call or all calls (depending on the design) have to go through the proxy application

Consensus Proposal

Before determining the consensus proposal, we have taken into account the following questions / considerations, extracted from the design options explained above:

- Do we use a JWT (to cache user identity) or just plain OIDC (and have to look up user info each time if stateless, which is inefficient, but uses only interoperable standards)?
    - If using JWT, do we keep it separate from OIDC Access Token, or combine into it? Separate requires clients to implement proprietary JWT support. Combining tokens requires a compatible IdP, which might require bundling Keycloak.
    - Do we choose a middle ground of requiring a compatible IdP that supports plain OIDC but with Access Token being JWTs?
    - Or can JWT be optional (detected by filter) and we fall back to plain OIDC w/ lookups?
- OIDC auth as a “public client” (from SPA) or “confidential client” (from backend)? If “confidential” it is more secure, but clients need to implement proprietary JWT support and the extent of support for this in backend libraries is currently unknown.
- Can we require that each installation have an OIDC IdP? Or should we bundle Keycloak: for SAML compatibility, and for ease of provisioning internal users, and to store additional user attributes?
- Can we burden the community with another mandatory component like Keycloak in addition to Postgres + Solr? (Which basically rules out going with Option 4 & 5 above or not)

Initial phase

- Make OIDC required, so the authentication is external to Dataverse
- Initially require an OIDC provider that provides JWT accessToken
- Includes a dev Keycloak setup and support for this OIDC mechanism in the Dataverse code. We would also use long-lived JWT access tokens for dev to simplify initial SPA work.

- Options exist to remove constraints on the OIDC provider, i.e. to allow shorter lived tokens, and/or to allow opaque accessTokens in the future. The community would be asked to help prioritize
- Support for SPA/current UI interop can be made with a flag that adds a session token upon OIDC login (example implementation of flags in PR #9230)

- The current Session API auth mechanism, available via feature flag, which makes the API accept the session cookie is a way to move forward even before this work is done.
-

Next phases

- Provide a ‘production ready’/secure implementation of keycloak for use by other clients against the current codebase

- (Production interop of SPA and current UI would require adding SameSite attributes on the session cookie and/or CSRF token - older browser may be a problem w.r.t. SameSite).

- Built-in users: - adding a login API call as in Option 1 lets people use them now (SPA or other clients with some custom logic), also compatible with current UI by adding a session cookie, longer term option is to move users to Keycloak.

Meeting notes

Meeting Dec 13th Notes

- Related eval docs from Oliver: https://iffmd.fz-juelich.de/-H_86K_-SEiFtjcLs-z-0Q#
- OIDC yes/no?
- Filter yes/no?
- If OIDC: ID token vs access token -> use access token, use if necessary to query userinfo
- treat local users separately?
- drop OIDC completely?
- Option 1: Session ID only approach
    - Might also be used for users coming from Shib, but JSF would have to stay for the time being.
    - Also would have to stay: all the underlying infrastructure, etc.
    - Low cost.
    - Lots of technical debt.
    - Need security tapes like CSRF.
    - Needs extended API support
- Option 2: OIDC only with HTTP Basic Auth (create and send session token)
    - Challenges with maintaining JSF and React login simultaneously.
    - SPA gets an id token and access token, used by SPA to call into API.
    - Dataverse can contact IdP to ask for more info about the user.
    - This is using a bearer token.
    - What about JSF?
        - JSF doesn't know anything about this.
        - We'd need to create a DataverseSession. Send a session cookie back to the browser.
        - Session cookie used in all future requests in case the user lands on a JSF page.
    - SPA would oversee the whole login process. No JSF when logging in.
    - Do we need to support JSF only?
    - We need to centralize the login on the SPA.
- Possible steps
    - SPA optional. Get session cookie so you can navigate to JSF pages.
    - Turn off JSF login. This could be done with a Apache rule.
- Johannes (via Oliver)
    - Make the API capable of accepting OIDC access tokens.
    - If you signed up with an OIDC provider, you get an access token.
    - Migration button. Reach out to OIDC provider. Same internal user based on unique identifier in database.
    - Developers would need to run an OIDC provider such as KeyCloak. Guillermo has been doing this with PKCE.

- How do we authenticate local users with OIDC?
    - Does Dataverse need to be an OIDC provider?
    - Is a non-OIDC JWT an option as an alternative to OIDC for local users? Where do the certificates come from? To validate? Refresh token? Manage lifecycle of token?
    - KeyCloak on dev machine. OpenIDAuthenticationDefinition in Payara -> Jakarta Security . In a separate library. In Security 3.0 as part of EE 10.
    - On login side, is the token coming from an external provider or ourselves?
    - HTTP Basic Auth to API? To get a real OIDC access token. Configure with local certificate only. Not full OIDC but could work. No need for KeyCloak for dev.
    - Allow API access via Basic Auth. SWORD already does this.
- final written proposal by sharing a doc on this channel by Jan 6th, 2023 (CoB)
- schedule the final discussion on Jan 11th
- How do we create a session id via API?
    - Come in with Basic Auth. You get a session token.
- When do these additions (creating and sending a session token etc.) go away?
    - When we no longer have any JSF pages.
- Should be configurable to turn off any of the API auth mechanisms
- API endpoints that already use the session (file download)
    - Add a filter? Cascade? Ends with Guest user?
- Option 3: Option 2 plus a ContainerRequestFilter and SecurityContext
- What's the mechanism for keeping the session cookie alive?
- The filter allows us to encapsulate JSF front end. When we remove JSF we can also remove the filter.
- No replay attacks using session cookie for a stateless REST API.
- Make download API calls no longer accept session id? (Would need migration of JSF part to use a different mechanism to get there - API tokens or other secrets should not be included in URLs as dangerous with copy&paste)