Overview

This implementation guide describes foundational patterns based on OAuth 2.0 for client applications to authorize, authenticate, and integrate with FHIR-based data systems.

Discovery of Server Capabilities and Configuration

SMART defines a discovery document available at `.well-known/smart-configuration` relative to a FHIR Server Base URL, allowing clients to learn the authorization endpoint URLs and features a server supports. This information helps client direct authorization requests to the right endpoint, and helps clients construct an authorization request that the server can support.

SMART Defines Two Patterns For Client *Authorization*

Authorization via **SMART App Launch**

Authorizes a user-facing client application ("App") to connect to a FHIR Server. This pattern allows for "launch context" such as *currently selected patient* to be shared with the app, based on a user's session inside an EHR or other health data software, and allows for delegation of a user's permissions to the app itself.

Authorization via **SMART Backend Services**

Authorizes a headless or automated client application ("Backend Service") to connect to a FHIR Server. This pattern allows for backend services to connect and interact with an EHR when there is no user directly involved in the launch process, or in other circumstances where permissions are assigned to the client out-of-band.

SMART Defines Two Patterns For Client *Authentication*

When clients need to authenticate, this implementation guide defines two methods.

Note that client authentication is not required in all authorization scenarios, and not all SMART clients are capable of authenticating (see discussion of "Public Clients" in the SMART App Launch overview).

Asymmetric ("private key JWT") authentication

Authenticates a client using an asymmetric keypair. This is SMART's preferred authentication method because it avoids sending a shared secret over the wire.

Symmetric ("client secret") authentication

Authenticate a client using a secret that has been pre-shared between the client and server.

Scopes for Limiting Acess

SMART uses a language of "scopes" to define specific access permissions that can be delegated to a client application. These scopes draw on FHIR API definitions for interactions, resource types, and search paramters to describe a permissions model. For example, an app might be granted scopes like `user/Encounter.rs`, allowing it to read and search for Encounters that are accessible to the user who has authorized the app. Similarly, a backend service might be granted scopes like `system/Encounter.rs`, allowing it to read and search for Encounters within the overall set of data it is configured to access. User-facing apps can also receive "launch context" to indicate details about the current patient or other aspects of a user’s EHR session or a user’s selections when launching the app.

Note that the scope syntax has changed since SMARTv1. Details are at Scopes for requesting clinical data.

Token Introspection

SMART defines a Token Introspection API allowing Resource Servers or software components to understand the scopes, users, patients, and other context associated with access tokenes. This pattern allows a looseer coupling between Resource Servers and Authorization Servers.

App Launch

The SMART App Launch Framework connects third-party applications to Electronic Health Record data, allowing apps to launch from inside or outside the user interface of an EHR system. The framework supports apps for use by clinicians, patients, and others via a PHR or Patient Portal or any FHIR system where a user can launch an app. It provides a reliable, secure authorization protocol for a variety of app architectures, including apps that run on an end-user’s device as well as apps that run on a secure server. The Launch Framework supports four key use cases:

1. Patients apps that launch standalone
2. Patient apps that launch from a portal
3. Provider apps that launch standalone
4. Provider apps that launch from a portal

These use cases support apps that perform data visualization, data collection, clinical decision support, data sharing, case reporting, and many other functions.

Profile audience and scope

This profile is intended to be used by developers of apps that need to access user identity information or other FHIR resources by requesting authorization from OAuth 2.0 compliant authorization servers. It is compatible with FHIR R2 (DSTU2) and later; this publication includes explicit definitions for FHIR R4.

OAuth 2.0 authorization servers are configured to mediate access based on a set of rules configured to enforce institutional policy, which may include requesting end-user authorization. This profile does not dictate the institutional policies that are implemented in the authorization server.

The profile defines a method through which an app requests authorization to access a FHIR resource, and then uses that authorization to retrieve the resource. Synchronization of patient context is not addressed; for use cases that require context synchronization (e.g., learning about when the in-context patient changes within an EHR session) see FHIRcast. In other words, if the patient chart is changed during the session, the application will not inherently be updated. Other security mechanisms, such as those mandated by HIPAA in the US (end-user authentication, session time-out, security auditing, and accounting of disclosures) are outside the scope of this profile.

This profile provides a mechanism to *delegate* an entity's permissions (e.g., a user's permissions) to a 3rd-party app. The profile includes mechanisms to delegate a limited subset of an entity's permissions (e.g., only sharing access to certain data types). However, this profile does not model the permissions that the entity has in the first place (e.g., it provides no mechanism to specify that a given entity should or should not be able to access specific records in an EHR). Hence, this profile is designed to work on top of an EHR's existing user and permissions management system, enabling a standardized mechanism for delegation.

Security and Privacy Considerations

App protection

The app is responsible for protecting itself from potential misbehaving or malicious values passed to its redirect URL (e.g., values injected with executable code, such as SQL) and for protecting authorization codes, access tokens, and refresh tokens from unauthorized access and use. The app developer must be aware of potential threats, such as malicious apps running on the same platform, counterfeit authorization servers, and counterfeit resource servers, and implement countermeasures to help protect both the app itself and any sensitive information it may hold. For background, see the OAuth 2.0 Threat Model and Security Considerations.

- Apps SHALL ensure that sensitive information (authentication secrets, authorization codes, tokens) is transmitted ONLY to authenticated servers, over TLS-secured channels.
- Apps SHALL generate an unpredictable `state` parameter for each user session; SHALL include `state` with all authorization requests; and SHALL validate the `state` value for any request sent to its redirect URL.
- An app SHALL NOT execute untrusted user-supplied inputs as code.
- An app SHALL NOT forward values passed back to its redirect URL to any other arbitrary or user-provided URL (a practice known as an "open redirector").
- An app SHALL NOT store bearer tokens in cookies that are transmitted as clear text.
- Apps SHOULD persist tokens and other sensitive data in app-specific storage locations only, and SHOULD NOT persist them in system-wide-discoverable locations.

Support for "public" and "confidential" apps

Within this profile we differentiate between the two types of apps defined in the OAuth 2.0 specification: confidential and public. The differentiation is based upon whether the execution environment within which the app runs enables the app to protect secrets. Pure client-side apps (for example, HTML5/JS browser-based apps, iOS mobile apps, or Windows desktop apps) can provide adequate security, but they may be unable to "keep a secret" in the OAuth2 sense. In other words, any "secret" key, code, or string that is statically embedded in the app can potentially be extracted by an end-user or attacker. Hence security for these apps cannot depend on secrets embedded at install-time.

For strategies and best practices to protecting a client secret refer to:

- OAuth 2.0 Threat Model and Security Considerations: 4.1.1. Threat: Obtaining Client Secrets
- OAuth 2.0 for Native Apps: 8.5. Client Authentication
- OAuth 2.0 Dynamic Client Registration Protocol

Use the confidential app profile if your app is *able* to protect a secret

for example:

- App runs on a trusted server with only server-side access to the secret
- App is a native app that uses additional technology (such as dynamic client registration and universal `redirect_uris`) to protect the secret

Use the public app profile if your app is *unable* to protect a secret

for example:

- App is an HTML5 or JS in-browser app (including single-page applications) that would expose the secret in user space
- App is a native app that can only distribute a secret statically

Considerations for PKCE Support

All SMART apps SHALL support Proof Key for Code Exchange (PKCE). PKCE is a standardized, cross-platform technique for clients to mitigate the threat of authorization code interception or injection. PKCE is described in IETF RFC 7636. SMART servers SHALL support the `S256` `code_challenge_method` and SHALL NOT support the `plain` method.

Related reading

Implementers can review the OAuth Security Topics guidance from IETF as a collection of Best Current Practices.

Some resources shared with apps following this IG may be considered Patient Sensitive; implementers should review the Core FHIR Specification's Security Page for additional security and privacy considerations.

SMART authorization & FHIR access: overview

An app can launch from within an existing EHR or Patient Portal session; this is known as an EHR launch. Alternatively, it can launch as a standalone app.

In an EHR launch, an opaque handle to the EHR context is passed along to the app as part of the launch URL. The app later will include this context handle as a request parameter when it requests authorization to access resources. Note that the complete URLs of all apps approved for use by users of this EHR will have been registered with the EHR authorization server.

Alternatively, in a standalone launch, when the app launches from outside an EHR session, the app can request context from the EHR authorization server during the authorization process described below.

Once an app receives a launch request, it requests authorization to access a FHIR resource by causing the browser to navigate to the EHR's authorization endpoint. Based on pre-defined rules and possibly end-user authorization, the EHR authorization server either grants the request by returning an authorization code to the app's redirect URL, or denies the request. The app then exchanges the authorization code for an access token, which the app presents to the EHR's resource server to access requested FHIR resources. If a refresh token is returned along with the access token, the app may use this to request a new access token, with the same scope, once the access token expires.

Top-level steps for SMART App Launch

1. Register App with EHR (*one-time step*, can be out-of-band)
2. Launch App: Standalone Launch or EHR Launch
3. Retrieve .well-known/smart-configuration
4. Obtain authorization code
5. Obtain access token
6. Access FHIR API
7. Refresh access token

AppAppEHR with Authorization ServerEHR with Authorization ServerFHIR ServerFHIR Serveropt[Precondition: Client Registration](may be out of band)alt[EHR Launch]EHR userlaunches appLaunch request[Standalone Launch]App userconnects to EHRDiscovery requestDiscovery responseAuthorization requestoptEHR incorporates user inputinto authorization decisionalt[Granted]Authorization grantedAccess token requestAccess token responseRequest Resources[Denied]Authorization error

Register App with EHR

Note: this is a one-time setup step, and can occur out-of-band.

Before a SMART app can run against an EHR, the app must be registered with that EHR's authorization service. SMART does not specify a standards-based registration process, but we encourage EHR implementers to consider the OAuth 2.0 Dynamic Client Registration Protocol for an out-of-the-box solution.

Request

No matter how an app registers with an EHR's authorization service, at registration time **every SMART app must**:

- Register zero or more fixed, fully-specified launch URL with the EHR's authorization server
- Register one or more fixed, fully-specified `redirect_uri`s with the EHR's authorization server. Note: In the case of native clients following the OAuth 2.0 for Native Apps specification (RFC 8252), it may be appropriate to leave the port as a dynamic variable in an otherwise fixed redirect URI.

For confidential clients, additional registration-time requirements are defined based on the client authentication method.

- For asymmetric client authentication: a JSON Web Key Set or JWSK URL is established
- For symmetric client authentication: a client secret is established

Response

The EHR confirms the app's registration parameters and communicates a `client_id` to the app.

Launch App: Standalone Launch

In SMART's standalone launch flow, a user selects an app from outside the EHR, for example by tapping an app icon on a mobile phone home screen.

Request

There is no explicit request associated with this step of the SMART App Launch process. The app proceeds to the next step of the SMART App Launch flow.

Response

N/A

Examples

- Public client
- Confidential client, asymmetric authentication
- Confidential client, symmetric authentication

Launch App: EHR Launch

In SMART's EHR launch flow, a user has established an EHR session, and then decides to launch an app. This could be a single-patient app (which runs in the context of a patient record), or a user-level app (like an appointment manager or a population dashboard).

Request

The EHR initiates a "launch sequence" by opening a new browser instance (or `iframe`) pointing to the app's registered launch URL and passing some context.

The following parameters are included:

Parameters

`iss`	required	Identifies the EHR's FHIR endpoint, which the app can use to obtain additional details about the E

`launch`	require d	Opaque identifier for this specific launch, and any EHR context associated with it. This parameter by passing along a `launch` parameter (see example below).

For example

A launch might cause the browser to navigate to:

```
Location: https://app/launch?iss=https%3A%2F%2Fehr%2Ffhir&launch=xyz123
```

On receiving the launch notification, the app would query the issuer's .well-known/smart-configuration endpoint which contains (among other details) the EHR's identifying the OAuth `authorize` and `token` endpoint URLs for use in requesting authorization to access FHIR resources.

Later, when the app prepares its authorization request, it includes `launch` as a requested scope and includes a `launch={launch id}` URL parameter, echoing the value it received from the EHR in this notification.

Response

The app proceeds to the next step of the SMART App Launch flow.

Retrieve `.well-known/smart-configuration`

In order to obtain launch context and request authorization to access FHIR resources, the app discovers the EHR FHIR server's SMART configuration metadata, including OAuth `authorization_endpoint` and `token_endpoint` URLs.

Request

The discovery URL is constructed by appending `.well-known/smart-configuration` to the FHIR Base URL. The app issues an HTTP GET to the discovery URL with an `Accept` header supporting `application/json`.

Response

The EHR responds with a SMART configuration JSON document as described in conformance

Examples

- Example request and response

Obtain authorization code

To proceed with a launch, the app constructs a request for an authorization code.

Request

The app supplies the following parameters to the EHR's "authorize" endpoint.

Note on PKCE Support: the EHR SHALL ensure that the `code_verifier` is present and valid when the code is exchanged for an access token.

Parameters

`response_type`	required	Fixed value: `code`.
`client_id`	required	The client's identifier.
`redirect_uri`	required	Must match one of the client's pre-registered redirect URIs.

`Launch`	conditional	When using the EHR Launch flow, this must match the launch value received from
`Scope`	required	Must describe the access that the app needs, including scopes like `patient/*.r` identity) and either: ● a `launch` value indicating that the app wants to receive already-establis ● a set of launch context requirements in the form `launch/patient`, which See [SMART on FHIR Access Scopes](#) details.
`State`	required	An opaque value used by the client to maintain state between the request and redirecting the user-agent back to the client. The parameter SHALL be used for attacks. The app SHALL use an unpredictable value for the state parameter wit random uuid is suitable).
`aud`	required	URL of the EHR resource server from which the app wishes to retrieve FHIR dat a counterfeit resource server. (Note: in the case of an EHR launch flow, this `aud` v the `aud` parameter is semantically equivalent to the `resource` parameter define we have decided not to rename it for reasons of backwards compatibility. We m implementer feedback indicates that alignment would be valuable. For the curr support a `resource` parameter as a synonym for `aud`.
`code_challenge`	required	This parameter is generated by the app and used for the code challenge, as spe when `code_challenge_method` is `'S256'`, this is the S256 hashed version of the c
`code_challenge_method`	required	Method used for the `code_challenge` parameter. Example value: `S256`. See [cons](#)

The app SHOULD limit its requested scopes to the minimum necessary (i.e., minimizing the requested data categories and the requested duration of access).

If the app needs to authenticate the identify of or retrieve information about the end-user, it should include two OpenID Connect scopes: `openid` and `fhirUser`. When these scopes are requested, and the request is granted, the app will receive an id_token along with the access token. For full details, see [SMART launch context parameters](#).

The following requirements are adopted from [OpenID Connect Core 1.0 Specification section 3.1.2.1](#):

- Authorization Servers SHALL support the use of the HTTP GET and POST methods at the Authorization Endpoint.
- Clients SHALL use either the HTTP GET or the HTTP POST method to send the Authorization Request to the Authorization Server. If using the HTTP GET method, the request parameters are serialized using URI Query String Serialization. If using the HTTP POST method, the request parameters are serialized using Form Serialization and the application/x-www-form-urlencoded content type.

For example

If an app needs demographics and observations for a single patient, and also wants information about the current logged-in user, the app can request:

- `patient/Patient.rs`
- `patient/Observation.rs`
- `openid fhirUser`

If the app was launched from an EHR, the app adds a `launch` scope and a `launch={launch id}` URL parameter, echoing the value it received from the EHR to be associated with the EHR context of this launch notification.

Apps using the standalone launch flow won't have a `launch` id at this point. These apps can declare launch context requirements by adding specific scopes to the authorization request: for example, `launch/patient` to indicate that the app needs a patient ID, or `launch/encounter` to indicate it needs an encounter. The EHR's "authorize" endpoint will take care of acquiring the context it needs (making it available to the app).
For example, if your app needs patient context, the EHR may provide the end-user with a patient selection widget. For full details, see [SMART launch context parameters](#).

The app then causes the browser to navigate the browser to the EHR's **authorization URL** as determined above. For example to cause the browser to issue a `GET`:

```
Location: https://ehr/authorize?
            response_type=code&
            client_id=app-client-id&
            redirect_uri=https%3A%2F%2Fapp%2Fafter-auth&
            launch=xyz123&

scope=launch+patient%2FObservation.rs+patient%2FPatient.rs+openid+fhirUser&
            state=98wrghuwuogerg97&
            aud=https://ehr/fhir
```

Alternatively, the following example shows one way for a client app to cause the browser to issue a `POST`, using HTML and javascript:

```
<html>
  <body onload="javascript:document.forms[0].submit()">
    <form method="post" action="https://ehr/authorize">
      <input type="hidden" name="response_type" value="code"/>
      <input type="hidden" name="client_id" value="app-client-id"/>
      <input type="hidden" name="redirect_uri" value="https://app/after-auth"/>
      <input type="hidden" name="launch" value="xyz123"/>
      <input type="hidden" name="scope" value="launch patient/Observation.rs
patient/Patient.rs openid fhirUser"/>
      <input type="hidden" name="state" value="98wrghuwuogerg97"/>
      <input type="hidden" name="aud" value="https://ehr/fhir"/>
    </form>
  </body>
</html>
```

Response

The authorization decision is up to the EHR authorization server, which may request authorization from the end-user. The EHR authorization server will enforce access rules based on local policies and optionally direct end-user input.

The EHR decides whether to grant or deny access. This decision is communicated to the app when the EHR authorization server returns an authorization code (or, if denying access, an error response). Authorization codes are short-lived, usually expiring within around one minute. The code is sent when the EHR authorization server causes the browser to navigate to the app's `redirect_uri`, with the following URL parameters:

Parameters

`code`	required	The authorization code generated by the authorization server. The authorization code *must* exp
`state`	required	The exact value received from the client.

The app SHALL validate the value of the state parameter upon return to the redirect URL and SHALL ensure that the state value is securely tied to the user's current session (e.g., by relating the state value to a session identifier issued by the app).

For example

Based on the `client_id`, current EHR user, configured policy, and perhaps direct user input, the EHR makes a decision to approve or deny access. This decision is communicated to the app by causing the browser to navigate to the app's registered `redirect_uri`. For example:

```
Location: https://app/after-auth?
  code=123abc&
  state=98wrghuwuogerg97
```

Examples

- Public client
- Confidential client, asymmetric authentication
- Confidential client, symmetric authentication

Obtain access token

After obtaining an authorization code, the app trades the code for an access token.

Request

The app issues an HTTP `POST` to the EHR authorization server's token endpoint URL, using content-type `application/x-www-form-urlencoded`, as described in section 4.1.3 of RFC6749.

For public apps, authentication is not possible (and thus not required), since a client with no secret cannot prove its identity when it issues a call. (The end-to-end system can still be secure because the client comes from a known, https protected endpoint specified and enforced by the redirect uri.) For confidential apps, authentication is required. Confidential clients SHOULD use Asymmetric Authentication if available, and MAY use Symmetric Authentication.

Parameters

`grant_type`	required	Fixed value: `authorization_code`
`code`	required	Code that the app received from the authorization server
`redirect_uri`	required	The same redirect_uri used in the initial authorization request
`code_verifier`	required	This parameter is used to verify against the `code_challenge` parameter previously
`client_id`	conditional	Required for public apps. Omit for confidential apps.

Response

The EHR authorization server SHALL return a JSON object that includes an access token or a message indicating that the authorization request has been denied. The JSON structure includes the following parameters:

Parameters

`access_token`	required	The access token issued by the authorization server
`token_type`	required	Fixed value: `Bearer`
`expires_in`	recommended	Lifetime in seconds of the access token, after which the token SHALL NOT be a
`scope`	required	Scope of access authorized. Note that this can be different from the scopes rec
`id_token`	optional	Authenticated user identity and user details, if requested
`refresh_token`	optional	Token that can be used to obtain a new access token, using the same or a subs

In addition, if the app was launched from within a patient context, parameters to communicate the context values MAY BE included. For example, a parameter like `"patient": "123"` would indicate the FHIR resource https://[fhir-base]/Patient/123. Other context parameters may also be available. For full details see SMART launch context parameters.

The parameters are included in the entity-body of the HTTP response, as described in section 5.1 of RFC6749.

The access token is a string of characters as defined in RFC6749 and RFC6750. The token is essentially a private message that the authorization server passes to the FHIR Resource Server, telling the FHIR server that the "message bearer" has been authorized to access the specified resources.
Defining the format and content of the access token is left up to the organization that issues the access token and holds the requested resource.

The authorization server's response SHALL include the HTTP "Cache-Control" response header field with a value of "no-store," as well as the "Pragma" response header field with a value of "no-cache."

The EHR authorization server decides what `expires_in` value to assign to an access token and whether to issue a refresh token, as defined in section 1.5 of RFC6749, along with the access token. If the app receives a refresh token along with the access token, it can exchange this refresh token for a new access token when the current access token expires (see step 5 below).

Apps SHOULD store tokens in app-specific storage locations only, not in system-wide-discoverable locations. Access tokens SHOULD have a valid lifetime no greater than one hour. Confidential clients may be issued longer-lived tokens than public clients.

A large range of threats to access tokens can be mitigated by digitally signing the token as specified in RFC7515 or by using a Message Authentication Code (MAC) instead. Alternatively, an access token can contain a reference to authorization information, rather than encoding the information directly into the token itself.
To be effective, such references must be infeasible for an attacker to guess. Using a reference may require an extra interaction between the resource server and the authorization server; the mechanics of such an interaction are not defined by this specification.

At this point, **the authorization flow is complete**. Follow steps below to work with data and refresh access tokens, as shown in the following sequence diagram.

Examples

- Public client

- Confidential client, asymmetric authentication
- Confidential client, symmetric authentication

Access FHIR API

With a valid access token, the app can access protected EHR data by issuing a FHIR API call to the FHIR endpoint on the EHR's resource server.

Request

From the access token resopnse, an app has received an OAuth2 bearer-type access token (`access_token` property) that can be used to fetch clinical data. The app issues a request that includes an `Authorization` header that presents the `access_token` as a "Bearer" token:

```
Authorization: Bearer {{access_token}}
```

(Note that in a real request, `{{access_token}}` is replaced with the actual token value.)

Response

The resource server SHALL validate the access token and ensure that it has not expired and that its scope covers the requested resource. The resource server also validates that the `aud` parameter associated with the authorization (see above) matches the resource server's own FHIR endpoint. The method used by the EHR to validate the access token is beyond the scope of this specification but generally involves an interaction or coordination between the EHR's resource server and the authorization server.

On occasion, an app may receive a FHIR resource that contains a "reference" to a resource hosted on a different resource server. The app SHOULD NOT blindly follow such references and send along its access_token, as the token may be subject to potential theft. The app SHOULD either ignore the reference, or initiate a new request for access to that resource.

Example Request and Response

Example

```
GET https://ehr/fhir/Patient/123

Authorization: Bearer i8hweunweunweofiwweoijewiwe
```

Response

```
{

  "resourceType": "Patient",

  "birthTime": ...

}
```

Refresh access token

Refresh tokens are issued to enable sessions to last longer than the validity period of an access token. The app can use the `expires_in` field from the token response (see step 5) to determine when its access token will expire. EHR implementers are also encouraged to consider using the OAuth 2.0 Token Introspection Protocol to provide an introspection endpoint that clients can use to examine the validity and meaning of tokens. An app with "online access" can continue to get new access tokens as long as the end-user remains online. Apps with "offline access" can continue to get new access tokens without the user being interactively engaged for cases where an application should have long-term access extending beyond the time when a user is still interacting with the client.

The app requests a refresh token in its authorization request via the `online_access` or `offline_access` scope (see SMART on FHIR Access Scopes for details). A server can decide which client types (public or confidential) are eligible for offline access and able to receive a refresh token. If granted, the EHR supplies a refresh_token in the token response. A refresh token SHALL BE bound to the same `client_id` and SHALL contain the same, or a subset of, the set of claims authorized for the access token with which it is associated. After an access token expires, the app requests a new access token by providing its refresh token to the EHR's token endpoint.]

Request

An HTTP `POST` transaction is made to the EHR authorization server's token URL, with content-type `application/x-www-form-urlencoded`. The decision about how long the refresh token lasts is determined by a mechanism that the server chooses. For clients with online access, the goal is to ensure that the user is still online.

- For public apps, authentication is not possible (and thus not required). For confidential apps, see authentication considerations in step 5.

The following request parameters are defined:

Parameters

`grant_type`	required	Fixed value: `refresh_token`.
`refresh_token`	required	The refresh token from a prior authorization response
`scope`	optional	The scopes of access requested. If present, this value must be a strict sub-set of the scop obtained at refresh time). A missing value indicates a request for the same scopes grante

Response

The response is a JSON object containing a new access token, with the following claims:

JSON Object property name

`access_token`	required	New access token issued by the authorization server.
`token_type`	required	Fixed value: bearer
`expires_in`	required	The lifetime in seconds of the access token. For example, the value 3600 denotes that the response was generated.
`scope`	required	Scope of access authorized. Note that this will be the same as the scope of the original ac by the app.
`refresh_token`	optional	The refresh token issued by the authorization server. If present, the app should discard ar replacing it with this new value.

In addition, if the app was launched from within a patient context, parameters to communicate the context values MAY BE included. For example, a parameter like `"patient": "123"` would indicate the FHIR resource https://[fhir-base]/Patient/123. Other context parameters may also be available. For full details see SMART launch context parameters.

Examples

- Public client
- Confidential client, asymmetric authentication
- Confidential client, symmetric authentication

2,1 Example App Launch for Public Client

- Example: App Launch with Asymmetric Authentication
    - Launch App
    - Retrieve .well-known/smart-configuration
    - Obtain authorization code
    - Retrieve access token
    - Access FHIR API
    - Refresh access token

2.1 Example: App Launch with Asymmetric Authentication

Launch App

This is a user-driven stepm triggering the subsequent workflow.

In this example, the launch is initiated aginst a FHIR server with a base URL of:

```
https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N2
EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/fhir
```

… and the app’s redirect URL has been registered as:

```
https://sharp-lake-word.glitch.me/graph.html
```

… and the app has been registered as a public client, assigned a `client_id` of:

```
demo_app_whatever
```

Retrieve .well-known/smart-configuration

```
curl -s
'https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/fhir/.well-known/smart-configuration
' \
  -H 'accept: application/json'


{
  "authorization_endpoint":
"https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/auth/authorize",
  "token_endpoint":
"https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/auth/token",
  "introspection_endpoint":
"https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/auth/introspect",
  "code_challenge_methods_supported": [
    "S256"
  ],
  "grant_types_supported": [
    "authorization_code"
  ],
  "token_endpoint_auth_methods_supported": [
    "private_key_jwt",
    "client_secret_basic"
  ],
  "token_endpoint_auth_signing_alg_values_supported": [
    "RS384",
    "ES384"
  ],
  "scopes_supported": [
    "openid",
    "fhirUser",
    "launch",
    "launch/patient",
    "patient/*.cruds"
    "user/*.cruds",
    "offline_access"
  ],
  "response_types_supported": [
    "code"
  ],
  "capabilities": [
    "launch-ehr",
    "launch-standalone",
    "client-public",
    "client-confidential-symmetric",
    "client-confidential-asymmetric",
    "context-passthrough-banner",
    "context-passthrough-style",
    "context-ehr-patient",
    "context-ehr-encounter",
    "context-standalone-patient",
    "context-standalone-encounter",
    "permission-offline",
    "permission-patient",
    "permission-user",
    "permission-v2",
    "authorize-post"
  ]
}
```

Obtain authorization code

Generate a PKCE code challenge and verifier, then redirect browser to the `authorize_endpoint` from the discovery response (newlines added for clarity):

```
https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N2
EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/auth/authorize?
  response_type=code&
  client_id=demo_app_whatever&

scope=launch%2Fpatient%20patient%2FObservation.rs%20patient%2FPatient.rs%20offline_acc
ess&
  redirect_uri=https%3A%2F%2Fsharp-lake-word.glitch.me%2Fgraph.html&

aud=https%3A%2F%2Fsmart.argo.run%2Fv%2Fr4%2Fsim%2FeyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJq
IjoiMSIsImIiOiI4N2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ%2Ffhir&state=0hJc1S
9O4oW54XuY&
  code_challenge=YPXe7B8ghKrj8PsT4L6ltupgI12NQJ5vblB07F4rGaw&
  code_challenge_method=S256
```

Receive authorization code when EHR redirects the browser back to (newlines added for clarity):

```
https://sharp-lake-word.glitch.me/graph.html?

code=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp7Im5lZWRfcGF0aWVudF9iYW5uZXIi
OnRydWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnby5ydW4vL3NtYXJ0LXN0eWxlLmpzb2
4iLCJwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU2YWFiM2M2In0sImNsaWVudF9pZCI6
ImRlbW9fYXBwX3doYXRldmVyIiwiY29kZV9jaGFsbGVuZ2VfbWV0aG9kIjoiUzI1NiIsImNvZGVfY2hhbGxlbm
dlIjoiWVBYZTdCOGdoS3JqOFBzVDRMNmx0dXBnSTEyTlFKNXZibEIwN0Y0ckdhdyIsInNjb3BlIjoibGF1bmNo
L3BhdGllbnQgcGF0aWVudC9PYnNlcnZhdGlvbi5ycyBwYXRpZW50L1BhdGllbnQucnMiLCJyZWRpcmVjdF91cm
kiOiJodHRwczovL3NoYXJwLWxha2Utd29yZC5nbGl0Y2gubWUvZ3JhcGguaHRtbCIsImlhdCI6MTYzMzUzMjAx
NCwiZXhwIjoxNjMzNTMyMzE0fQ.xilM68Bavtr9IpklYG-j96gTxAda9r4Z_boe2zv3A3E&
  state=0hJc1S9O4oW54XuY
```

Retrieve access token

Generate a client authentication assertion and prepare arguments for POST to token API (newlines added for clarity):

```
client_id=demo_app_whatever&
```

```
code=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp7Im5lZWRfcGF0aWVudF9iYW5uZXIi
OnRydWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnby5ydW4vL3NtYXJ0LXN0eWxlLmpzb2
4iLCJwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU2YWFiM2M2In0sImNsaWVudF9pZCI6
ImRlbW9fYXBwX3doYXRldmVyIiwiY29kZV9jaGFsbGVuZ2VfbWV0aG9kIjoiUzI1NiIsImNvZGVfY2hhbGxlbm
dlIjoiWVBYZTdCOGdoS3JqOFBzVDRMNmx0dXBnSTEyTlFKNXZibEIwN0Y0ckdhdyIsInNjb3BlIjoibGF1bmNo
L3BhdGllbnQgcGF0aWVudC9PYnNlcnZhdGlvbi5ycyBwYXRpZW50L1BhdGllbnQucnMiLCJyZWRpcmVjdF91cm
kiOiJodHRwczovL3NoYXJwLWxha2Utd29yZC5nbGl0Y2gubWUvZ3JhcGguaHRtbCIsImlhdCI6MTYzMzUzMjAx
NCwiZXhwIjoxNjMzNTMyMzE0fQ.xilM68Bavtr9IpklYG-j96gTxAda9r4Z_boe2zv3A3E&

grant_type=authorization_code&

redirect_uri=https%3A%2F%2Fsharp-lake-word.glitch.me%2Fgraph.html&

code_verifier=o28xyrYY7-lGYfnKwRjHEZWlFIPlzVnFPYMWbH-g_BsNnQNem-IAg9fDh92X0KtvHCPO5_C-
RJd2QhApKQ-2cRp-S_W3qmTidTEPkeWyniKQSF9Q_k10Q5wMc8fGzoyF
```

Issue POST to the token endpoint:

```
curl
'https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/auth/token' \

  -H 'accept: application/json' \

  -H 'content-type: application/x-www-form-urlencoded' \

  --data-raw
'client_id=demo_app_whatever&code=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp
7Im5lZWRfcGF0aWVudF9iYW5uZXIiOnRydWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnb
y5ydW4vL3NtYXJ0LXN0eWxlLmpzb24iLCJwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU
2YWFiM2M2In0sImNsaWVudF9pZCI6ImRlbW9fYXBwX3doYXRldmVyIiwiY29kZV9jaGFsbGVuZ2VfbWV0aG9kI
joiUzI1NiIsImNvZGVfY2hhbGxlbmdlIjoiWVBYZTdCOGdoS3JqOFBzVDRMNmx0dXBnSTEyTlFKNXZibEIwN0Y
0ckdhdyIsInNjb3BlIjoibGF1bmNoL3BhdGllbnQgcGF0aWVudC9PYnNlcnZhdGlvbi5ycyBwYXRpZW50L1Bhd
GllbnQucnMiLCJyZWRpcmVjdF91cmkiOiJodHRwczovL3NoYXJwLWxha2Utd29yZC5nbGl0Y2gubWUvZ3JhcGg
uaHRtbCIsImlhdCI6MTYzMzUzMjAxNCwiZXhwIjoxNjMzNTMyMzE0fQ.xilM68Bavtr9IpklYG-j96gTxAda9r
4Z_boe2zv3A3E&grant_type=authorization_code&redirect_uri=https%3A%2F%2Fsharp-lake-word
.glitch.me%2Fgraph.html&code_verifier=o28xyrYY7-lGYfnKwRjHEZWlFIPlzVnFPYMWbH-g_BsNnQNe
m-IAg9fDh92X0KtvHCPO5_C-RJd2QhApKQ-2cRp-S_W3qmTidTEPkeWyniKQSF9Q_k10Q5wMc8fGzoyF'




{

  "need_patient_banner": true,

  "smart_style_url": "https://smart.argo.run/smart-style.json",

  "patient": "87a339d0-8cae-418e-89c7-8651e6aab3c6",

  "token_type": "Bearer",

  "scope": "launch/patient patient/Observation.rs patient/Patient.rs",

  "expires_in": 3600,

  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuZWVkX3BhdGllbnRfYmFubmVyIjp0cnVlLCJzbWFydF9z
dHlsZV91cmwiOiJodHRwczovL3NtYXJ0LmFyZ28ucnVuLy9zbWFydC1zdHlsZS5qc29uIiwicGF0aWVudCI6Ij
g3YTMzOWQwLThjYWUtNDE4ZS04OWM3LTg2NTFlNmFhYjNjNiIsInRva2VuX3R5cGUiOiJiZWFyZXIiLCJzY29w
ZSI6ImxhdW5jaC9wYXRpZW50IHBhdGllbnQvT2JzZXJ2YXRpb24ucnMgcGF0aWVudC9QYXRpZW50LnJzIiwiY2
xpZW50X2lkIjoiZGVtb19hcHBfd2hhdGV2ZXIiLCJleHBpcmVzX2luIjozNjAwLCJpYXQiOjE2MzM1MzIwMTQs
ImV4cCI6MTYzMzUzNTYxNH0.PzNw23IZGtBfgpBtbIczthV2hGwanG_eyvthVS8mrG4",

  "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp7Im5lZWRfcGF0aWVudF9iYW5uZXIiOnRy
dWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnby5ydW4vL3NtYXJ0LXN0eWxlLmpzb24iLC
JwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU2YWFiM2M2In0sImNsaWVudF9pZCI6ImRl
bW9fYXBwX3doYXRldmVyIiwic2NvcGUiOiJsYXVuY2gvcGF0aWVudCBwYXRpZW50L09ic2VydmF0aW9uLnJzIH
BhdGllbnQvUGF0aWVudC5ycyBvZmZsaW5lX2FjY2VzcyIsImlhdCI6MTYzMzUzMzg1OSwiZXhwIjoxNjY1MDY5
ODU5fQ.Q41QwZCEQlZ16M7YwvYuVbUP03mRFJoqRxL8SS8_ImM"

}
```

Access FHIR API

```
curl
'https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/fhir/Observation?code=4548-4&_sort%3
Adesc=date&_count=10&patient=87a339d0-8cae-418e-89c7-8651e6aab3c6' \

  -H 'accept: application/json' \

  -H 'authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuZWVkX3BhdGllbnRfYmFubmVyIjp0cnVlLCJzbWFydF9zd
HlsZV91cmwiOiJodHRwczovL3NtYXJ0LmFyZ28ucnVuLy9zbWFydC1zdHlsZS5qc29uIiwicGF0aWVudCI6Ijg
3YTMzOWQwLThjYWUtNDE4ZS04OWM3LTg2NTFlNmFhYjNjNiIsInRva2VuX3R5cGUiOiJiZWFyZXIiLCJzY29wZ
SI6ImxhdW5jaC9wYXRpZW50IHBhdGllbnQvT2JzZXJ2YXRpb24ucnMgcGF0aWVudC9QYXRpZW50LnJzIiwiY2x
pZW50X2lkIjoiZGVtb19hcHBfd2hhdGV2ZXIiLCJleHBpcmVzX2luIjozNjAwLCJpYXQiOjE2MzM1MzIwMTQsI
mV4cCI6MTYzMzUzNTYxNH0.PzNw23IZGtBfgpBtbIczthV2hGwanG_eyvthVS8mrG4'


{

  "resourceType": "Bundle",

  "id": "9e3ed23b-b62e-4a3d-9ac8-9b66a67f700d",

  "meta": {

    "lastUpdated": "2021-10-06T10:52:52.847-04:00"

  },

  "type": "searchset",

  "total": 11,

  "link": [

    {

      "relation": "self",

      "url":
"https://smart.argo.run/v/r4/fhir/Observation?_count=10&_sort%3Adesc=date&code=4548-4&
patient=87a339d0-8cae-418e-89c7-8651e6aab3c6"

    },

    {

      "relation": "next",

      "url":
"https://smart.argo.run/v/r4/fhir?_getpages=9e3ed23b-b62e-4a3d-9ac8-9b66a67f700d&_getp
agesoffset=10&_count=10&_pretty=true&_bundletype=searchset"

    }

  ],

  "entry": [

    {
```

```
<SNIPPED for brevity>
```

Refresh access token

Generate a client authentication assertion and prepare arguments for POST to token API (newlines added for clarity)

```
client_id=demo_app_whatever&

grant_type=refresh_token&

refresh_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp7Im5lZWRfcGF0aWVudF9
iYW5uZXIiOnRydWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnby5ydW4vL3NtYXJ0LXN0e
WxlLmpzb24iLCJwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU2YWFiM2M2In0sImNsaWV
udF9pZCI6ImRlbW9fYXBwX3doYXRldmVyIiwic2NvcGUiOiJsYXVuY2gvcGF0aWVudCBwYXRpZW50L09ic2Vyd
mF0aW9uLnJzIHBhdGllbnQvUGF0aWVudC5ycyBvZmZsaW5lX2FjY2VzcyIsImlhdCI6MTYzMzUzMzg1OSwiZXh
wIjoxNjY1MDY5ODU5fQ.Q41QwZCEQlZ16M7YwvYuVbUP03mRFJoqRxL8SS8_ImM&

curl
'https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N
2EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/auth/token' \

  -H 'accept: application/json' \

  -H 'content-type: application/x-www-form-urlencoded' \

  --data-raw
'client_id=demo_app_whatever&code=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp
7Im5lZWRfcGF0aWVudF9iYW5uZXIiOnRydWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnb
y5ydW4vL3NtYXJ0LXN0eWxlLmpzb24iLCJwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU
2YWFiM2M2In0sImNsaWVudF9pZCI6ImRlbW9fYXBwX3doYXRldmVyIiwiY29kZV9jaGFsbGVuZ2VfbWV0aG9kI
joiUzI1NiIsImNvZGVfY2hhbGxlbmdlIjoieFFzdkN5c2FMbEZvVkU5ZV92MTFiWmNwUlR6RW5wVnIzY2c2VTJ
YeFpFbyIsInNjb3BlIjoibGF1bmNoL3BhdGllbnQgcGF0aWVudC9PYnNlcnZhdGlvbi5ycyBwYXRpZW50L1Bhd
GllbnQucnMiLCJyZWRpcmVjdF91cmkiOiJodHRwczovL3NoYXJwLWxha2Utd29yZC5nbGl0Y2gubWUvZ3JhcGg
uaHRtbCIsImlhdCI6MTYzMzUzMzY1NCwiZXhwIjoxNjMzNTMzOTU0fQ.ovs8WkW7ViCvoiTGJXxWb21OtiJfUm
wgXwkt3a1gNRc&grant_type=authorization_code'


{

  "need_patient_banner": true,

  "smart_style_url": "https://smart.argo.run/smart-style.json",

  "patient": "87a339d0-8cae-418e-89c7-8651e6aab3c6",

  "token_type": "Bearer",

  "scope": "launch/patient patient/Observation.rs patient/Patient.rs offline_access",

  "expires_in": 3600,

  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuZWVkX3BhdGllbnRfYmFubmVyIjp0cnVlLCJzbWFydF9z
dHlsZV91cmwiOiJodHRwczovL3NtYXJ0LmFyZ28ucnVuLy9zbWFydC1zdHlsZS5qc29uIiwicGF0aWVudCI6Ij
g3YTMzOWQwLThjYWUtNDE4ZS04OWM3LTg2NTFlNmFhYjNjNiIsInJlZnJlc2hfdG9rZW4iOiJleUpoYkdjaU9p
SklVekkxTmlJc0luUjVjQ0k2SWtwWFZDSjkuZXlKamIyNTBaWGgwSWpwN0ltNWxaV1JmY0dGMGFXVnVkRjlpWV
c1dVpYSWlPblJ5ZFdVc0luTnRZWEowWDNOMGVXeGxYM1Z5YkNJNkltaDBkSEJ6T2k4dmMyMWhjblF1WVhKbmJ5
NXlkVzR2TDNOdFlYSjBMWE4wZVd4bExtcHpiMjRpTENKd1lYUnBaVzUwSWpvaU9EZGhNek01WkRBdE9HTmhaUz
AwTVRobExUZzVZemN0T0RZMU1XVTJZV0ZpTTJNMkluMHNJbU5zYVdWdWRGOXBaQ0k2SW1SbGJXOWZZWEJ3WDNk
b1lYUmxkbVZ5SWl3aWMyTnZjR1VpT2lKc1lYVnVZMmd2Y0dGMGFXVnVkQ0J3WVhScFpXNTBMMDlpYzJWeWRtRj
BhVzl1TG5KeklIQmhkR2xsYm5RdlVHRjBhV1Z1ZEM1eWN5QnZabVpzYVc1bFgyRmpZMlZ6Y3lJc0ltbGhkQ0k2
TVRZek16VXpNemcxT1N3aVpYaHdJam94TmpZMU1EWTVPRFU1ZlEuUTQxUXdaQ0VRbFoxNk03WXd2WXVWYlVQMD
NtUkZKb3FSeEw4U1M4X0ltTSIsInRva2VuX3R5cGUiOiJiZWFyZXIiLCJzY29wZSI6ImxhdW5jaC9wYXRpZW50
IHBhdGllbnQvT2JzZXJ2YXRpb24ucnMgcGF0aWVudC9QYXRpZW50LnJzIG9mZmxpbmVfYWNjZXNzIiwiY2xpZW
50X2lkIjoiZGVtb19hcHBfd2hhdGV2ZXIiLCJleHBpcmVzX2luIjozNjAwLCJpYXQiOjE2MzM1MzM4NTksImV4
cCI6MTYzMzUzNzQ1OX0.-4vtO6iADkH7HM6-IqoSchEMv2mVsztjHg-5RBkPXrc",

  "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJjb250ZXh0Ijp7Im5lZWRfcGF0aWVudF9iYW5uZXIiOnRy
dWUsInNtYXJ0X3N0eWxlX3VybCI6Imh0dHBzOi8vc21hcnQuYXJnby5ydW4vL3NtYXJ0LXN0eWxlLmpzb24iLC
JwYXRpZW50IjoiODdhMzM5ZDAtOGNhZS00MThlLTg5YzctODY1MWU2YWFiM2M2In0sImNsaWVudF9pZCI6ImRl
bW9fYXBwX3doYXRldmVyIiwic2NvcGUiOiJsYXVuY2gvcGF0aWVudCBwYXRpZW50L09ic2VydmF0aW9uLnJzIH
BhdGllbnQvUGF0aWVudC5ycyBvZmZsaW5lX2FjY2VzcyIsImlhdCI6MTYzMzUzMzg1OSwiZXhwIjoxNjY1MDY5
ODU5fQ.Q41QwZCEQlZ16M7YwvYuVbUP03mRFJoqRxL8SS8_ImM"

}
```

2.2 Example App Launch for Asymmetric Client Auth

2.3 Example: App Launch with Asymmetric Authentication

Launch App

This is a user-driven stepm triggering the subsequent workflow.

In this example, the launch is initiated aginst a FHIR server with a base URL of:

```
https://smart.argo.run/v/r4/sim/eyJtIjoiMSIsImsiOiIxIiwiaSI6IjEiLCJqIjoiMSIsImIiOiI4N2
EzMzlkMC04Y2FlLTQxOGUtODljNy04NjUxZTZhYWIzYzYifQ/fhir
```

… and the app’s redirect URL has been registered as:

```
https://sharp-lake-word.glitch.me/graph.html
```

… and the app’s public key has been registered as:

```
{
  "kty": "EC",
  "crv": "P-384",
  "x": "wcE8O55ro6aOuTf5Ty1k_IG4mTcuLiVercHouge1G5Ri-leevhev4uJzlHpi3U8r",
  "y": "mLRgz8Giu6XA_AqG8bywqbygShmd8jowflrdx0KQtM5X4s4aqDeCRfcpexykp3aI",
  "kid": "afb27c284f2d93959c18fa0320e32060",
  "alg": "ES384",
}
```

(For reproducibility: the corresponding private key parameter `"d"` is `"WcrTiYk8jbI-Sd1sKNpqGmELWGG08bf_y9SSlnC4cpAl5GRdHHN9gKYlPvMFqiJ5"`. This would not be shared in a real-world registration scenario.)

… and the app has been assigned a `client_id` of:

```
demo_app_whatever
```

3 Backend Services

Profile Audience and Scope

This profile is intended to be used by developers of backend services (clients) that autonomously (or semi-autonomously) need to access resources from FHIR servers that have pre-authorized defined scopes of access. This specification handles use cases complementary to the SMART App Launch protocol. Specifically, this profile describes the runtime process by which the client acquires an access token that can be used to retrieve FHIR resources. This specification is designed to work with FHIR Bulk Data Access, but is not restricted to use for retrieving bulk data; it may be used to connect to any FHIR API endpoint, including both synchronous and asynchronous access.

Use this profile when the following conditions apply:

- The target FHIR authorization server can register the client and pre-authorize access to a defined set of FHIR resources.
- The client may run autonomously, or with user interaction that does not include access authorization.
- The client supports `client-confidential-asymmetric` authentication
- No compelling need exists for a user to authorize the access at runtime.

Note See Also: The FHIR specification includes a set of security considerations including security, privacy, and access control. These considerations apply to diverse use cases and provide general guidance for choosing among security specifications for particular use cases.

Examples

- An analytics platform or data warehouse that periodically performs a bulk data import from an electronic health record system for analysis of a population of patients.
- A lab monitoring service that determines which patients are currently admitted to the hospital, reviews incoming laboratory results, and generates clinical alerts when specific trigger conditions are met. Note that in this example, the monitoring service may be a backend client to multiple servers.
- A data integration service that periodically queries the EHR for newly registered patients and synchronizes these with an external database
- A utilization tracking system that queries an EHR every minute for bed and room usage and displays statistics on a wall monitor.
- Public health surveillance studies that do not require real-time exchange of data.

Underlying Standards

- HL7 FHIR RESTful API
- RFC5246, The Transport Layer Security Protocol, V1.2

- RFC6749, The OAuth 2.0 Authorization Framework
- RFC7515, JSON Web Signature
- RFC7517, JSON Web Key
- RFC7518, JSON Web Algorithms
- RFC7519, JSON Web Token (JWT)
- RFC7521, Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants
- RFC7523, JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
- RFC7591, OAuth 2.0 Dynamic Client Registration Protocol

Conformance Language

This specification uses the conformance verbs SHALL, SHOULD, and MAY as defined in RFC2119. Unlike RFC 2119, however, this specification allows that different applications may not be able to interoperate because of how they use optional features. In particular:

1. SHALL: an absolute requirement for all implementations
2. SHALL NOT: an absolute prohibition against inclusion for all implementations
3. SHOULD/SHOULD NOT: A best practice or recommendation to be considered by implementers within the context of their particular implementation; there may be valid reasons to ignore an item, but the full implications must be understood and carefully weighed before choosing a different course
4. MAY: This is truly optional language for an implementation; can be included or omitted as the implementer decides with no implications

Top-level steps for Backend Services Authorization

Backend ServiceBackend ServiceFHIR authorization serverFHIR authorization serverFHIR resource serverFHIR resource serveropt[Precondition: Client Registration](may be out of band)Discovery requestDiscovery responseAccess token requestalt[Granted]Access token responseRequest Resources[Denied]Authorization error

1. Register Backend Service (*one-time step*, can be out-of-band)
2. Retrieve .well-known/smart-configuration
3. Obtain access token
4. Access FHIR API

Register SMART Backend Service (communicating public keys)

Before a SMART client can run against a FHIR server, the client SHALL register with the server by following the registration steps described in `client-confidential-asymmetric` authentication.

Retrieve `.well-known/smart-configuration`

In order to request authorization to access FHIR resources, the app discovers the EHR FHIR server's SMART configuration metadata, including OAuth `token` endpoint URL.

Request

The app issues an HTTP GET with an `Accept` header supporting `application/json` to retrieve the SMART configuration file.

Response

Servers respond with a discovery response that meets discovery requirements described in `client-confidential-asymmetric` authentication.

Example Request and Response

For a full example, see example request and response.

Obtain acess token

By the time a client has been registered with the FHIR authorization server, the key elements of organizational trust will have been established. That is, the client will be considered "pre-authorized" to access FHIR resources. Then, at runtime, the client will need to obtain an access token in order to retrieve FHIR resources as pre-authorized. Such access tokens are issued by the FHIR authorization server, in accordance with the OAuth 2.0 Authorization Framework, RFC6749.

Because the authorization scope is limited to protected resources previously arranged with the FHIR authorization server, the client credentials grant flow, as defined in Section 4.4 of RFC6749, may be used to request authorization. Use of the client credentials grant type requires that the client SHALL be a "confidential" client capable of protecting its authentication credential.

This specification describes requirements for requesting an access token through the use of an OAuth 2.0 client credentials flow, with a JWT assertion as the client's authentication mechanism. The exchange, as depicted below, allows the client to authenticate itself to the FHIR authorization server and to request a short-lived access token in a single exchange.

Request

To begin the exchange, the client SHALL use the Transport Layer Security (TLS) Protocol Version 1.2 (RFC5246) or a more recent version of TLS to authenticate the identity of the FHIR authorization server and to establish an encrypted, integrity-protected link for securing all exchanges between the client and the FHIR authorization server's token endpoint. All exchanges described herein between the client and the FHIR server SHALL be secured using TLS V1.2 or a more recent version of TLS .

Before a client can request an access token, it generates a one-time-use authentication JWT as described in `client-confidential-symmetric` authentication. After generating this authentication JWT, the client requests an access token via HTTP `POST` to the FHIR authorization server's token endpoint URL, using content-type `application/x-www-form-urlencoded` with the following parameters:

Parameters

`scope`	required	The scope of access requested. See note about scopes be
`grant_type`	required	Fixed value: `client_credentials`
`client_assertion_type`	required	Fixed value: `urn:ietf:params:oauth:client-assertion-typ`
`client_assertion`	required	Signed authentication JWT value (see above)

Scopes

The client is pre-authorized by the server: at registration time or out of band, it is given the authority to access certain data. The client then includes a set of scopes in the access token request, which causes the server to apply additional access restrictions following the SMART Scopes syntax. For Backend Services, requested scopes will be `system/` scopes (for example `system/Observation.rs`, which requests an access token capable of reading all Observations that the client has been pre-authorized to access).

Response

Enforce Authorization

There are several cases where a client might ask for data that the server cannot or will not return:

- Client explicitly asks for data that it is not authorized to see (e.g. a client asks for Observation resources but has scopes that only permit access to Patient resources). In this case a server SHOULD respond with a failure to the initial request.
- Client explicitly asks for data that the server does not support (e.g., a client asks for Practitioner resources but the server does not support FHIR access to Practitioner data). In this case a server SHOULD respond with a failure to the initial request.
- Client explicitly asks for data that the server supports and that appears consistent with its access scopes – but some additional out-of-band rules/policies/restrictions prevents the client from being authorized to see these data. In this case, the server MAY withhold certain results from the response, and MAY indicate to the client that results were withheld by including OperationOutcome information in the "error" array for the response as a partial success.

Rules regarding circumstances under which a client is required to obtain and present an access token along with a request are based on risk-management decisions that each FHIR resource service needs to make, considering the workflows involved, perceived risks, and the organization's risk-management policies. Refresh tokens SHOULD NOT be issued.

Validate Authentication JWS

The FHIR authorization server validates a client's authentication JWT according to the `client-confidential-asymmetric` authentication profile. See JWT validation rules.

Evaluate Requested Access

Once the client has been authenticated, the FHIR authorization server SHALL mediate the request to assure that the scope requested is within the scope pre-authorized to the client.

Issue Access Token

If an error is encountered during the authorization process, the FHIR authorization server SHALL respond with the appropriate error message defined in Section 5.2 of the OAuth 2.0 specification. The FHIR authorization server SHOULD include an `error_uri` or `error_description` as defined in OAuth 2.0.

If the access token request is valid and authorized, the FHIR authorization server SHALL issue an access token in response. The access token response SHALL be a JSON object with the following properties:

Access token response: property names

`access_token`	required	The access token issued by the FHIR authorization server.
`token_type`	required	Fixed value: `bearer`.
`expires_in`	required	The lifetime in seconds of the access token. The recommended value is `300`, for a f
`scope`	required	Scope of access authorized. Note that this can be different from the scopes reques

To minimize risks associated with token redirection, the scope of each access token SHOULD encompass, and be limited to, the resources requested. Access tokens issued under this profile SHALL be short-lived; the `expires_in` value SHOULD NOT exceed `300`, which represents an expiration-time of five minutes.

Example Token Request and Response

For a full example, see example token request and response.

Access FHIR API

With a valid access token, the app can access protected FHIR data by issuing a FHIR API call to the FHIR endpoint on the FHIR resource server.

Request

From the access token resopnse, an app has received an OAuth2 bearer-type access token (`access_token` property) that can be used to fetch clinical data. The app issues a request that includes an `Authorization` header that presents the `access_token` as a "Bearer" token:

```
Authorization: Bearer {{access_token}}
```

(Note that in a real request, `{{access_token}}` is replaced with the actual token value.)

Response

The resource server SHALL validate the access token and ensure that it has not expired and that its scope covers the requested resource. The method used by the EHR to validate the access token is beyond the scope of this specification but generally involves an interaction or coordination between the EHR's resource server and the authorization server.

On occasion, an Backend Service may receive a FHIR resource that contains a "reference" to a resource hosted on a different resource server. The Backend Service SHOULD NOT blindly follow such references and send along its access_token, as the token may be subject to potential theft. The Backend Service SHOULD either ignore the reference, or initiate a new request for access to that resource.

Example Request and Response

For a full example, see example FHIR API request and response.

4 Scopes and Launch Context

SMART on FHIR's authorization scheme uses OAuth scopes to communicate (and negotiate) access requirements. Providing apps with access to broad data sets is consistent with current common practices (e.g. interface engines also provide access to broad data sets); access is also limited based on the privileges of the user in context. In general, we use scopes for three kinds of data:

1. Clinical data
2. Contextual data
3. Identity data

Launch context is a negotiation where a client asks for specific launch context parameters (e.g. `launch/patient`). A server can decide which launch context parameters to provide, using the client's request as an input into the decision process. When granting a patient-level scopes like `patient/*.rs`, the server SHALL provide a "patient" launch context parameter.

Quick Start

Here is a quick overview of the most commonly used scopes. Read on below for complete details.

Scope	Grants

`patient/*.rs`	Permission to read and search any resource for the current patient (see notes on wildcard scopes below).
`user/*.cruds`	Permission to read and write all resources that the current user can access (see notes on wildcard scopes below).
`openid` `fhirUser`	Permission to retrieve information about the current logged-in user.
`launch`	Permission to obtain launch context when app is launched from an EHR.
`launch/patient`	When launching outside the EHR, ask for a patient to be selected at launch time.
`offline_access`	Request a `refresh_token` that can be used to obtain a new access token to replace an expired one, even after the end-user no longer is online after the access token expires.
`online_access`	Request a `refresh_token` that can be used to obtain a new access token to replace an expired one, and that will be usable for as long as the end-user remains online.

SMART's scopes are used to delegate access

SMART's scopes allow a client to request the delegation of a specific set of access rights; such rights are always limited by underlying system policies and permissions.

For example:

- If a client uses SMART App Launch to request `user/*.cruds` and is granted these scopes by a user, these scopes convey "full access" relative to the user's underlying permissions. If the underlying user has limited permissions, the client will face these same limitations.
- If a client uses SMART Backend Services to request `system/*.cruds`, these scopes convey "full access" relative to a pre-configured client policy. If the pre-configured policy imposes limited permissions, the client will face these same limitations.

Neither SMART on FHIR nor the FHIR Core specification provide a way to model the "underlying" permissions at play here; this is a lower-level responsibility in the access control stack. As such, clients can attempt to perform FHIR operations based on the scopes they are granted — but depending on the details of the underlying permission system (e.g., the permissions of the approving user and/or permissions assigned in a client-specific policy) these requests may be rejected, or results may be omitted from responses.

For instance, a client may receive:

- `200 OK` response to a search interaction that appears to be allowed by the granted scopes, but where results have been omitted from the response Bundle.
- `403 Forbidden` response to a write interaction that appears to be allowed by the granted scopes.

Applications reading may receive results that have been filtered or redacted based on the underlying permissions of the delegating authority, or may be refused access (see guidance at https://hl7.org/fhir/security.html#AccessDenied).

Scopes for requesting clinical data

SMART on FHIR defines OAuth2 access scopes that correspond directly to FHIR resource types. These scopes impact the access an application may have to FHIR resources (and actions). We define permissions to support the following FHIR REST API interactions:

- `c` for `create`
    - Type level create
- `r` for `read`
    - Instance level read

    - Instance level vread
    - Instance level history
- `u` for `update`
    - Instance level update Note that some servers allow for an update operation to create a new instance, and this is allowed by the update scope
    - Instance level patch
- `d` for `delete`
    - Instance level delete
- `s` for `search`
    - Type level search
    - Type level history
    - System level search
    - System level history

Valid suffixes are a subset of the in-order string `.cruds`. For example, to convey support for creating and updating observations, use scope `patient/Observation.cu`. To convey support for reading and searching observations, use scope `patient/Observation.rs`. For backwards compatibility with scopes defined in the SMART App Launch 1.0 specification, servers SHOULD advertise the `permission-v1` capability in their `.well-known/smart-configuration` discovery document, SHOULD return v1 scopes when v1 scopes are requested and granted, and SHOULD process v1 scopes with the following semantics in v2:

- v1 `.read` ⇒ v2 `.rs`
- v1 `.write` ⇒ v2 `.cud`
- v1 `.*` ⇒ v2 `.cruds`

Scope requests with undefined or out of order interactions MAY be ignored, replaced with server default scopes, or rejected. For example, a request of `.dus` is not a defined scope request. This policy is to prevent misinterpretation of scopes with other conventions (e.g., interpreting `.read` as `.rd` and granting extraneous delete permissions).

Batches and Transactions

SMART 2.0 does not define specific scopes for batch or transaction interactions. These system-level interactions are simply convience wrappers for other interactions. As such, batch and transaction requests should be validated based on the actual requests within them.

Scope Equivalence

Multiple scopes compounded or expanded are equivalent to each other. E.g., `Observation.rs` is interchangeable with `Observation.r Observation.s`. In order to reduce token size, it is recomended that scopes be factored to their shortest form.

Finer-grained resource constraints using search parameters

In SMART 1.0, scopes were based entirely on FHIR Resource types, as in `patient/Observation.read` (for Observations) or `patient.Immunization.read` (for Immunizations). In SMART 2.0, we provide more detailed constraints based on FHIR REST API search parameter syntax. To apply these constraints, add a query string suffix to existing scopes, starting with `?` and followed by a series of `param=value` items separated by `&`. For example, to request read and search access to laboratory observations but not other observations, the scope `patient/Observation.rs?category=http://terminology.hl7.org/CodeSystem/observation-category|laboratory`.

Requirements for support

We're seeking community consensus on a small common core of search parameters for broad support; we reserve the right to make some search parameters mandatory in the future.

Experimental features

Because the search parameter based syntax here is quite general, it opens up the possibility of using many features that servers may have trouble supporting in a consistent and performant fashion. Given the current level of implementation experience, the following features should be considered experimental, even if they are supported by a server:

- Use of search modifiers such as `Observation.rs?code:in=http://valueset.example.org/ValueSet/diabetes-codes`
- Use of search parameter chaining such as `Observation.rs?patient.birthdate=1990`
- Use of FHIR's `_filter` capabilities

Scope size over the wire

Scope strings appear over the wire at several points in an OAuth flow. Implementers should be aware that fine-grained controls can lead to a proliferation of scopes, increasing in the length of the `scope` string for app authorizations. As such, implementers should take care to avoid putting arbitrarily large scope strings in places where they might not "fit". The following considerations apply, presented in the sequential order of a SMART App Launch:

- When initiating an authorization request, app developers should prefer POST-based authorization requests to GET-based requests, since this avoid URL length limits that might apply to GET-based authorization requests. (For example, somme current-generation browsers have a 32kB length limit for values displayed in the URL bar.)
- In the authorization code redirect response, no scopes are included, so these considerations do not apply.
- In the access token response, no specific limits apply, since this payload comes in response to a client-initiated POST.
- In the token introspection response, no specific limits apply, since this payload comes in response to a client-initiated POST.
- In the access token itself, implementation-specific considerations may apply. SMART leaves access token formats out of scope, so formally there are no restrictions. But since access tokens are included in HTTP headers, servers should take care to ensure they do not get too large. For example, some current-generation HTTP servers have an 8kB limit on header length. To remain under this limit, authorization servers that use structured token formats like JWT might consider embedding handles or pointers to scopes, rather than embedding literal scopes in an access token. Alternatively, authorization servers might establish an internal convention mapping shorter scope names into longer scopes (or common combinations of longer scopes).

Clinical Scope Syntax

Expressed as a railroad diagram, the scope language is:

patientusersystem/FHIR Resource Type*.cruds?param=value&

Patient-specific scopes

Patient-specific scopes allow access to specific data about a single patient. *Which* patient is not specified here: clinical data scopes are all about *what* and not *who* which is handled in the next section. Patient-specific scopes start with `patient/`. Note that some EHRs may not enable access to all related resources - for example, Practitioners linked to/from Patient-specific resources. Note that if a FHIR server supports replacing one Patient record with another via `Patient.link`, the server documentation SHALL describe its authorization behavior.

Let's look at a few examples:

Goal	**Scope**	**Notes**
Read all observations about a patient	`patient/Observation.rs`	
Read demographics about a patient	`patient/Patient.r`	Note the difference in capitalization between "patient" the permission type and "Patient" the resource.

Add new blood pressure readings for a patient	`patient/Observation.c`	Note that the permission is broader than our goal: with this scope, an app can add not only blood pressures, but other observations as well. Note also that write access does not imply read access.
Read all available data about a patient	`patient/*.cruds`	See notes on wildcard scopes below.

User-level scopes

User-level scopes allow access to specific data that a user can access. Note that this isn't just data *about* the user; it's data *available to* that user. User-level scopes start with `user/`.

Let's look at a few examples:

Goal	**Scope**	**Notes**
Read a feed of all new lab observations across a patient population	`user/Observation.rs`	
Manage all appointments to which the authorizing user has access	`user/Appointment.cruds`	Individual attributes such as `d` for delete could be removed if not required.
Manage all resources on behalf of the authorizing user	`user/*.cruds`	
Select a patient	`user/Patient.rs`	Allows the client app to select a patient.

System-level scopes

System-level scopes describe data that a client system is directly authorized to access; these scopes are useful in cases where there is no user in the loop, such as a data monitoring or reporting service. System-level scopes start with `system/`.

Let's look at a few examples:

Goal	**Scope**	**Notes**
Alert engine to monitor all lab observations in a health system	`system/Observation.rs`	Read-only access to observations.
Perform bulk data export across all available data within a FHIR server	`system/*.rs`	Full read/search for all resources.
System-level bridge, turning a V2 ADT feed into FHIR Encounter resources	`system/Encounter.cud`	Write access to Encounters.

Wildcard scopes

As noted previously, clients can request clinical scopes that contain a wildcard (`*`) for the FHIR resource. When a wildcard is requested for the FHIR resource, the client is asking for all data for all available FHIR resources, both now *and in the future*. This is an important distinction to understand, especially for the entity responsible for granting authorization requests from clients.

For instance, imagine a FHIR server that today just exposes the Patient resource. The authorization server asking a patient to authorize a SMART app requesting `patient/*.cruds` should inform the user that they are being asked to grant this SMART app access to not just the currently accessible data about them (patient demographics), but also any additional data the FHIR server may be enhanced to expose in the future (eg, genetics).

As with any requested scope, the scopes ultimately granted by the authorization server may differ from the scopes requested by the client! When dealing with wildcard clinical scope requests, this is often true.

As a best practice, clients should examine the granted scopes by the authorization server and respond accordingly. Failure to do so may lead to situations in which the client attempts to access FHIR resources they were not granted access only to receieve an authorization failure by the FHIR server.

For example, imagine a client with the goal of obtaining read and write access to a patient's allergies and as such, requests the clinical scope of `patient/AllergyIntolerance.cruds`. The authorization server may respond in a variety of ways with respect to the scopes that are ultimately granted. The following table outlines several, but not an exhaustive list of scenarios for this example:

Granted Scope	**Notes**
`patient/AllergyIntolerance.cruds`	The client was granted exactly what it requested: patient-level read and write access to allergies via the same requested wildcard scope.
`patient/AllergyIntolerance.rs` `patient/AllergyIntolerance.cud`	The client was granted exactly what it requested: patient-level CRUDS access to allergies. However, note that this was communicated via two explicit scopes rather than a single scope.
`patient/AllergyIntolerance.rs`	The client was granted just patient-level read access to allergies.
`patient/AllergyIntolerance.cud`	The client was granted just patient-level write access to allergies.
`patient/*.rs`	The client was granted read access to all data on the patient.
`patient/*.cruds`	The client was granted its requested scopes as well as read/write access to all other data on the patient.
`patient/Observation.rs`	The client was granted an entirely different scope: patient-level read access to the patient's observations. While this behavior is unlikely for a production quality authorization server, this scenario is technically possible.
`""` (empty scope string – no scopes granted)	The authorization server chose to not grant any of the requested scopes.

As a best practice, clients are encouraged to request only the scopes and permissions they need to function and avoid the use of wildcard scopes purely for the sake of convenience. For instance, if your allergy management app requires patient-level read and write access to allergies, requesting the `patient/AllergyIntolerance.cruds` scope is acceptable. However, if your app only requires access to read allergies, requesting a scope of `patient/AllergyIntolerance.rs` would be more appropriate.

Scopes for requesting context data

These scopes affect what context parameters will be provided in the access token response. Many apps rely on contextual data from the EHR to answer questions like:

- Which patient record is currently "open" in the EHR?
- Which encounter is currently "open" in the EHR?

- At which clinic, hospital ward, or patient room is the end-user currently working?

To request access to such details, an app asks for "launch context" scopes in addition to whatever clinical access scopes it needs. Launch context scopes are easy to tell apart from clinical data scopes, because they always begin with `launch`.

There are two general approaches to asking for launch context data, depending on the details of how your app is launched.

Apps that launch from the EHR

Apps that launch from the EHR will be passed an explicit URL parameter called `launch`, whose value must associate the app's authorization request with the current EHR session. For example, If an app receives the URL parameter `launch=abc123`, then it requests the scope `launch` and provides an additional URL parameter of `launch=abc123`.

The application could choose to also provide `launch/patient` and/or `launch/encounter` as "hints" regarding which contexts the app would like the EHR to gather. The EHR MAY ignore these hints (for example, if the user is in a workflow where these contexts do not exist).

If an application requests a clinical scope which is restricted to a single patient (e.g. `patient/*.rs`), and the authorization results in the EHR is granting that scope, the EHR SHALL establish a patient in context. The EHR MAY refuse authorization requests including `patient/` that do not also include a valid `launch`, or it MAY infer the `launch/patient` scope.

Standalone apps

Standalone apps that launch outside the EHR do not have any EHR context at the outset. These apps must explicitly request EHR context. The EHR SHOULD provide the requested context if requested by the following scopes, unless otherwise noted:

Requesting context with scopes

Requested Scope	**Meaning**
`launch/patient`	Need patient context at launch time (FHIR Patient resource). See note below.
`launch/encounter`	Need encounter context at launch time (FHIR Encounter resource).
(Others)	This list can be extended by any SMART EHR if additional context is required. When specifying resource types, convert the type names to *all lowercase* (e.g. `launch/diagnosticreport`).

Note on `launch/patient`: If an application requests a clinical scope which is restricted to a single patient (e.g. `patient/*.rs`), and the authorization results in the EHR granting that scope, the EHR SHALL establish a patient in context. The EHR MAY refuse authorization requests including `patient/` that do not also include a valid `launch/patient` scope, or it MAY infer the `launch/patient` scope.

Launch context arrives with your `access_token`

Once an app is authorized, the token response will include any context data the app requested – along with (potentially!) any unsolicited context data the EHR decides to communicate. For example, EHRs may use launch context to communicate UX and UI expectations to the app (see `need_patient_banner` below).

Launch context parameters come alongside the access token. They will appear as JSON parameters:

```
{
  "access_token": "secret-xyz",
  "patient": "123",
  "fhirContext": ["DiagnosticReport/123", "Organization/789"],
//...
}
```

Here are the launch context parameters to expect:

Launch context parameter	Example value	Meaning
`patient`	`"123"`	String value with a patient id, indicating that the app was launched in the context of FHIR Patient 123. If the app has any patient-level scopes, they will be scoped to Patient 123.
`encounter`	`"123"`	String value with an encounter id, indicating that the app was launched in the context of FHIR Encounter 123.
`fhirContext`	`["Appointment/123"]`	Array of relative resource References to any resource type other than "Patient" or "Encounter". It is not prohibited to have more than one Reference to a given *type* of resource.
`need_patient_banner`	`true` or `false` (boolean)	Boolean value indicating whether the app was launched in a UX context where a patient banner is required (when `true`) or not required (when `false`). An app receiving a value of `false` should not take up screen real estate displaying a patient banner.
`intent`	`"reconcile-medications"`	String value describing the intent of the application launch (see notes below)
`smart_style_url`	`"https://ehr/styles/smart_v1.json"`	String URL where the EHR's style parameters can be retrieved (for apps that support styling)
`tenant`	`"2ddd6c3a-8e9a-44c6-a305-52111ad302a2"`	String conveying an opaque identifier for the healthcare organization that is launching the app. This parameter is intended primarily to support EHR Launch scenarios.

Notes on launch context parameters

`fhirContext`

`fhirContext`: To allow application flexibility, while also maintaining backwards compatibility (and to keep a predictable JSON structure), any contextual resource types (other than Patient and Encounter) that were requested by a launch scope will appear in this parameter. The Patient and Encounter resource types will *not be deprecated from top-level parameters*, and they will *not be permitted* within the `fhirContext` array.

App Launch Intent **(optional)**

`intent`: Some SMART apps might offer more than one context or user interface that can be accessed during the SMART launch. The optional `intent` parameter in the launch context provides a mechanism for the SMART EHR to communicate to the client app which specific context should be displayed as the outcome of the launch. This allows for closer integration between the EHR and client, so that different launch points in the EHR UI can target specific displays within the client app.

For example, a patient timeline app might provide three specific UI contexts, and inform the SMART EHR (out of band, at app configuration time) of the `intent` values that can be used to launch the app directly into one of the three contexts. The app might respond to `intent` values like:

- `summary-timeline-view` - A default UI context, showing a data summary
- `recent-history-timeline` - A history display, showing a list of entries
- `encounter-focused-timeline` - A timeline focused on the currently in-context encounter

If a SMART EHR provides a value that the client does not recognize, or does not provide a value, the client app SHOULD display a default application UI context.

Note: *SMART makes no effort to standardize `intent` values*. Intents simply provide a mechanism for tighter custom integration between an app and a SMART EHR. The meaning of intents must be negotiated between the app and the EHR.

SMART App Styling (experimental1)

`smart_style_url`: In order to mimic the style of the SMART EHR more closely, SMART apps can check for the existence of this launch context parameter and download the JSON file referenced by the URL value, if provided.

The URL should serve a “SMART Style” JSON object with one or more of the following properties:

```
{
  color_background: "#edeae3",
  color_error: "#9e2d2d",
  color_highlight: "#69b5ce",
  color_modal_backdrop: "",
  color_success: "#498e49",
  color_text: "#303030",
  dim_border_radius: "6px",
  dim_font_size: "13px",
  dim_spacing_size: "20px",
  font_family_body: "Georgia, Times, 'Times New Roman', serif",
  font_family_heading: "'HelveticaNeue-Light', Helvetica, Arial, 'Lucida Grande',
sans-serif;"
}
```

The URL value itself is to be considered a version key for the contents of the SMART Style JSON: EHRs must return a new URL value in the `smart_style_url` launch context parameter if the contents of this JSON is changed.

Style Property	**Description**
`color_background`	The color used as the background of the app.
`color_error`	The color used when UI elements need to indicate an area or item of concern or dangerous action, such as a button to be used to delete an item, or a display an error message.
`color_highlight`	The color used when UI elements need to indicate an area or item of focus, such as a button used to submit a form, or a loading indicator.

`color_modal_backdrop`	The color used when displaying a backdrop behind a modal dialog or window.
`color_success`	The color used when UI elements need to indicate a positive outcome, such as a notice that an action was completed successfully.
`color_text`	The color used for body text in the app.
`dim_border_radius`	The base corner radius used for UI element borders (0px results in square corners).
`dim_font_size`	The base size of body text displayed in the app.
`dim_spacing_size`	The base dimension used to space UI elements.
`font_family_body`	The list of typefaces to use for body text and elements.
`font_family_heading`	The list of typefaces to use for content heading text and elements.

SMART client apps that can adjust their styles should incorporate the above property values into their stylesheets, but are not required to do so.

Optionally, if the client app detects a new version of the SMART Style object (i.e. a new URL is returned the `smart_style_url` parameter), the client can store the new property values and request approval to use the new values from a client app stakeholder. This allows for safeguarding against poor usability that might occur from the immediate use of these values in the client app UI.

Scopes for requesting identity data

Some apps need to authenticate the end-user. This can be accomplished by requesting the scope `openid`. When the `openid` scope is requested, apps can also request the `fhirUser` scope to obtain a FHIR resource representation of the current user.

When these scopes are requested (and the request is granted), the app will receive an `id_token` that comes alongside the access token.

This token must be validated according to the OIDC specification. To learn more about the user, the app should treat the `fhirUser` claim as the URL of a FHIR resource representing the current user. This URL MAY be absolute (e.g., `https://ehr.example.org/Practitioner/123`), or it MAY be relative to the FHIR server base URL associated with the current authorization request (e.g., `Practitioner/123`). This will be a resource of type `Patient`, `Practitioner`, `RelatedPerson`, or `Person`. Note that the FHIR server base URL is the same as the URL represented in the `aud` parameter passed in to the authorization request. Note that `Person` is only used if the other resource types do not apply to the current user, for example, the "authorized representative" for >1 patients.

The OpenID Connect Core specification describes a wide surface area with many optional capabilities. To be considered compatible with the SMART's `sso-openid-connect` capability, the following requirements apply:

- Response types: The EHR SHALL support the Authorization Code Flow, with the request parameters as defined in SMART App Launch. Support is not required for parameters that OIDC lists as optional (e.g. `id_token_hint`, `acr_value`), but EHRs are encouraged to review these optional parameters.
- Public Keys Published as Bare JWK Keys: The EHR SHALL publish public keys as bare JWK keys (which MAY also be accompanied by X.509 representations of those keys).
- Claims: The EHR SHALL support the inclusion of SMART's `fhirUser` claim within the `id_token` issued for any requests that grant the `openid` and `fhirUser` scopes.
- Signed ID Token: The EHR SHALL support Signing ID Tokens with RSA SHA-256.
- A SMART app SHALL NOT pass the `auth_time` claim or `max_age` parameter to a server that does not support receiving them.

Note that support for the following features is optional:

- `claims` parameters on the authorization request
- Request Objects on the authorization request
- UserInfo endpoint with claims exposed to clients

Scopes for requesting a refresh token

To request a `refresh_token` that can be used to obtain a new access token after the current access token expires, add one of the following scopes:

Scope	Grants
`online_access`	Request a `refresh_token` that can be used to obtain a new access token to replace an expired one, and that will be usable for as long as the end-user remains online.
`offline_access`	Request a `refresh_token` that can be used to obtain a new access token to replace an expired token, and that will remain usable for as long as the authorization server and end-user will allow, regardless of whether the end-user is online.

Extensions

Additional context parameters and scopes can be used as extensions using the following namespace conventions:

- use a *full URI* that you control (e.g. http://example.com/scope-name)
- use any string starting with `__` (two underscores)

Example: Extra context - `fhirContext` for FHIR Resource References

EHR Launch

Suppose a SMART on FHIR server supports additional launch context during an EHR Launch, perhaps communicating the ID of an `ImagingStudy` that is open in the EHR at the time of app launch. The server could return an access token response where the `fhirContext` array includes a value such as `ImagingStudy/123`.

Standalone Launch

Suppose a SMART on FHIR server supports additional launch context during a Standalone Launch, perhaps providing an ability for the user to select an `ImagingStudy` during the launch. A client could request this behavior by requesting a `launch/imagingstudy` scope (note that launch requests scopes are always lower case); then after allowing the user to select an `ImagingStudy`, the server could return an access token response where the `fhirContext` array includes a value such as `ImagingStudy/123`.

Example: Extra context - extensions for non-FHIR context

Suppose a SMART on FHIR server wishes to communicate additional context, such as a custom “dark mode” flag, providing clients a hint about whether they should render a UI suitable for use in low-light environments. The EHR could accomplish this by returning an access token response where an extension property is present. The EHR could choose an extension property as a full URL (e.g., `{..., "https://ehr.example.org/props/dark-mode": true}`) or by using a `"__"` prefix (e.g., `{..., "__darkMode": true}`).

Example: Extra scopes - extensions for non-FHIR APIs

Suppose a SMART on FHIR server supports a custom behavior like allowing users to choose their own profile photos through a custom non-FHIR API. The server can designate a custom scope using a full URL (e.g., `https://ehr.example.org/scopes/profilePhoto.manage`) or by using a `"__"` prefix (e.g., `__profilePhoto.manage`) and associate this scope with the custom behavior. The server could advertise this scope in its developer-facing documentation, and also in the `scopes_supported` array of its `.well-known/smart-configuration` file. Clients requesting authorization could include this scope alongside other standardized scopes, so the `scope` parameter of the authorization request might look like: `launch/patient patient/*.rs __profilePhoto.manage`. If the user grants these scopes, the access token response would then include a `scope` value that matches the original request.

Steps for using an ID token

1. Examine the ID token for its "issuer" property
2. Perform a `GET {issuer}/.well-known/openid-configuration`
3. Fetch the server's JSON Web Key by following the "jwks_uri" property
4. Validate the token's signature against the public key from step #3
5. Extract the `fhirUser` claim and treat it as the URL of a FHIR resource

Worked examples

- Worked Python example: rendered

Appendix: URI representation of scopes

In some circumstances - for example, exchanging what scopes users are allowed to have, or sharing what they did choose), the scopes must be represented as URIs. When this is done, the standard URI is to prefix the SMART scopes with http://smarthealthit.org/fhir/scopes/, so that a scope would be `http://smarthealthit.org/fhir/scopes/patient/*.read`.

openID scopes have a URI prefix of http://openid.net/specs/openid-connect-core-1_0#5

Asymmetric Authentication

- Profile Audience and Scope
- Underlying Standards
- Conformance Language
- Advertising server support for this profile
- Registering a client (communicting public keys)
- Authenticating to the Token endpoint
- Worked example

Profile Audience and Scope

This profile desribes SMART's `client-confidential-asymmetric` authentication mechanism. It is intended for for SMART clients that can manage and sign assertions with asymmetric keys. Specifically, this profile describes the registration-time metadata required for a client using asymmetric keys, and the runtime process by which a client can authenticate to an OAuth server's token endpoint. This profile can be implemented by user-facing SMART apps in the context of the SMART App Launch flow or by SMART Backend Services that establish a connection with no user-facing authorization step.

Use this profile when the following conditions apply:

- The target FHIR authorization server supports SMART's `client-confidential-asymmetric` capability
- The client can maange asymmetric keys for authentication
- The client is able to protect a private key

Note See Also: The FHIR specification includes a set of security considerations including security, privacy, and access control. These considerations apply to diverse use cases and provide general guidance for choosing among security specifications for particular use cases.

Underlying Standards

- HL7 FHIR RESTful API
- RFC5246, The Transport Layer Security Protocol, V1.2
- RFC6749, The OAuth 2.0 Authorization Framework
- RFC7515, JSON Web Signature
- RFC7517, JSON Web Key

- RFC7518, JSON Web Algorithms
- RFC7519, JSON Web Token (JWT)
- RFC7521, Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants
- RFC7523, JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
- RFC7591, OAuth 2.0 Dynamic Client Registration Protocol

Conformance Language

This specification uses the conformance verbs SHALL, SHOULD, and MAY as defined in RFC2119. Unlike RFC 2119, however, this specification allows that different applications may not be able to interoperate because of how they use optional features. In particular:

1. SHALL: an absolute requirement for all implementations
2. SHALL NOT: an absolute prohibition against inclusion for all implementations
3. SHOULD/SHOULD NOT: A best practice or recommendation to be considered by implementers within the context of their particular implementation; there may be valid reasons to ignore an item, but the full implications must be understood and carefully weighed before choosing a different course
4. MAY: This is truly optional language for an implementation; can be included or omitted as the implementer decides with no implications

Advertising server support for this profile

As described in the Conformance section, a server advertises its support for SMART Confidential Clients with Asymmetric Keys by including the `client-confidential-asymmetric` capability at is `.well-known/smart-configuration` endpoint; configuration properties include `token_endpoint`, `scopes_supported`, `token_endpoint_auth_methods_supported` (with values that include `private_key_jwt`), and `token_endpoint_auth_signing_alg_values_supported` (with values that include at least one of `RS384`, `ES384`).

Example `.well-known/smart-configuration` Response

```
HTTP/1.1 200 OK
Content-Type: application/json


{
  "token_endpoint": "https://ehr.example.com/auth/token",
  "token_endpoint_auth_methods_supported": ["private_key_jwt"],
  "token_endpoint_auth_signing_alg_values_supported": ["RS384", "ES384"],
  "scopes_supported": ["system/*.rs"]
}
```

Registering a client (communicting public keys)

Before a SMART client can run against a FHIR server, the client SHALL generate or obtain an asymmetric key pair and SHALL register its public key set with that FHIR server’s authorization service (referred to below as the “FHIR authorization server”). SMART does not require a standards-based registration process, but we encourage FHIR service implementers to consider using the OAuth 2.0 Dynamic Client Registration Protocol.

No matter how a client registers with a FHIR authorization server, the client SHALL register the **public key** the client will use to authenticate itself to the FHIR authorization server. The public key SHALL be conveyed to the FHIR authorization server in a JSON Web Key (JWK) structure presented within a JWK Set, as defined in JSON Web Key Set (JWKS). The client SHALL protect the associated private key from unauthorized disclosure and corruption.

For consistency in implementation, FHIR authorization servers SHALL support registration of client JWKs using both of the following techniques (clients SHALL choose a server-supported method at registration time):

1. URL to JWK Set (strongly preferred). This URL communicates the TLS-protected endpoint where the client's public JWK Set can be found. This endpoint SHALL be accessible via TLS without authentication or authorization. Advantages of this approach are that it allows a client to rotate its own keys by updating the hosted content at the JWK Set URL, assures that the public key used by the FHIR authorization server is current, and avoids the need for the FHIR authorization server to maintain and protect the JWK Set. The client SHOULD return a "Cache-Control" header in its JWKS response

2. JWK Set directly (strongly discouraged). If a client cannot host the JWK Set at a TLS-protected URL, it MAY supply the JWK Set directly to the FHIR authorization server at registration time. In this case, the FHIR authorization server SHALL protect the JWK Set from corruption, and SHOULD remind the client to send an update whenever the key set changes. Conveying the JWK Set directly carries the limitation that it does not enable the client to rotate its keys in-band. Including both the current and successor keys within the JWK Set helps counter this limitation. However, this approach places increased responsibility on the FHIR authorization server for protecting the integrity of the key(s) over time, and denies the FHIR authorization server the opportunity to validate the currency and integrity of the key at the time it is used.

The client SHALL be capable of generating a JSON Web Signature in accordance with RFC7515. The client SHALL support both `RS384` and `ES384` for the JSON Web Algorithm (JWA) header parameter as defined in RFC7518. The FHIR authorization server SHALL be capable of validating signatures with at least one of `RS384` or `ES384`. Over time, best practices for asymmetric signatures are likely to evolve. While this specification mandates a baseline of support clients and servers MAY support and use additional algorithms for signature validation. As a reference, the signature algorithm discovery protocol `token_endpoint_auth_signing_alg_values_supported` property is defined in OpenID Connect as part of the OAuth2 server metadata.

No matter how a JWK Set is communicated to the FHIR authorization server, each JWK SHALL represent an asymmetric key by including `kty` and `kid` properties, with content conveyed using "bare key" properties (i.e., direct base64 encoding of key material as integer values). This means that:

- For RSA public keys, each JWK SHALL include `n` and `e` values (modulus and exponent)
- For ECDSA public keys, each JWK SHALL include `crv`, `x`, and `y` values (curve, x-coordinate, and y-coordinate, for EC keys)

Upon registration, the client SHALL be assigned a `client_id`, which the client SHALL use when requesting an access token.

Authenticating to the Token endpoint

This specification describes how a client authenticates using an asymmetric key, e.g. when requesting an access token duringSMART App LaunchSMART Backend Services

Authentication is based on the OAuth 2.0 client credentials flow, with a JWT assertion as the client's authentication mechanism.

To begin the exchange, the client SHALL use the Transport Layer Security (TLS) Protocol Version 1.2 (RFC5246) or a more recent version of TLS to authenticate the identity of the FHIR authorization server and to establish an encrypted, integrity-protected link for securing all exchanges between the client and the FHIR authorization server's token endpoint. All exchanges described herein between the client and the FHIR server SHALL be secured using TLS V1.2 or a more recent version of TLS .

Request

Before a client can request an access token, it SHALL generate a one-time-use JSON Web Token (JWT) that will be used to authenticate the client to the FHIR authorization server. The authentication JWT SHALL include the following claims, and SHALL be signed with the client's private key (which SHOULD be an `RS384` or `ES384` signature). For a practical reference on JWT, as well as debugging tools and client libraries, see https://jwt.io.

Authentication JWT Header Values

`alg`	require d	The JWA algorithm (e.g., `RS384`, `ES384`) used for signing the authentication JWT.
`kid`	require d	The identifier of the key-pair used to sign this JWT. This identifier SHALL be unique within the client's
`typ`	require d	Fixed value: `JWT`.
`jku`	optional	The TLS-protected URL to the JWK Set containing the public key(s) accessible without authentication URL value that the client supplied to the FHIR authorization server at client registration time. When a JWK Set URL or the JWK Set supplied at registration time. See Signature Verification for details.

Authentication JWT Claims

`iss`	require d	Issuer of the JWT -- the client's `client_id`, as determined during registration with the FHIR authoriza the `sub` claim)
`sub`	require d	The client's `client_id`, as determined during registration with the FHIR authorization server (note th
`aud`	require d	The FHIR authorization server's "token URL" (the same URL to which this authentication JWT will be
`exp`	require d	Expiration time integer for this authentication JWT, expressed in seconds since the "Epoch" (1970-01 minutes in the future.
`jti`	require d	A nonce string value that uniquely identifies this authentication JWT.

After generating an authentication JWT, the client requests an access token following either the SMART App Launch or the SMART Backend Services specification. Authentication details are conveyed using the following additional properties on the token request:

Parameters

`client_assertion_type`	required	Fixed value: `urn:ietf:params:oauth:client-assertion-typ`
`client_assertion`	required	Signed authentication JWT value (see above)

Response

Signature Verification

The FHIR authorization server SHALL validate the JWT according to the processing requirements defined in Section 3 of RFC7523 including validation of the signature on the JWT.

In addition, the authentication server SHALL:

- check that the `jti` value has not been previously encountered for the given `iss` within the maximum allowed authentication JWT lifetime (e.g., 5 minutes). This check prevents replay attacks.
- ensure that the `client_id` provided is known and matches the JWT's `iss` claim

To resolve a key to verify signatures, a FHIR authorization server SHALL follow this algorithm:

1. If the `jku` header is present, verify that the `jku` is whitelisted (i.e., that it matches the JWKS URL value supplied at registration time for the specified `client_id`).
   a. If the `jku` header is not whitelisted, the signature verification fails.
   b. If the `jku` header is whitelisted, create a set of potential keys by dereferencing the `jku` URL. Proceed to step 3.
2. If the `jku` header is absent, create a set of potential key sources consisting of all keys found in the registration-time JWKS or found by dereferencing the registration-time JWK Set URL. Proceed to step 3.
3. Identify a set of candidate keys by filtering the potential keys to identify the single key where the `kid` matches the value supplied in the client's JWT header, and the `kty` is consistent with the signature algorithm supplied in the client's JWT header (e.g., `RSA` for a JWT using an RSA-based signature, or `EC` for a JWT using an EC-based signature). If no keys match, or more than one key matches, the verification fails.
4. Attempt to verify the JWK using the key identified in step 3.

To retrieve the keys from a JWKS URL in step 1 or step 2, a FHIR authorization server issues a HTTP GET request that URL to obtain a JWKS response. For example, if a client has registered a JWKS URL of https://client.example.com/path/to/jwks.json, the server retrieves the client's JWKS with a GET request for that URL, including a header of `Accept: application/json`.

If an error is encountered during the authentication process, the server SHALL respond with an `invalid_client` error as defined by the OAuth 2.0 specification.

- The FHIR authorization server SHALL NOT cache a JWKS for longer than the client's cache-control header indicates.
- The FHIR authorization server SHOULD cache a client's JWK Set according to the client's cache-control header; it doesn't need to retrieve it anew every time.

Processing of the access token request proceeds according to either the SMART App Launch or the SMART Backend Services specification.

Worked example

Assume that a "bilirubin result monitoring service" client has registered with a FHIR authorization server whose token endpoint is at "https://authorize.smarthealthit.org/token", establishing the following

- JWT "issuer" URL: `https://bili-monitor.example.com`
- OAuth2 `client_id`: `bili_monitor`
- JWK identfier: `kid` value (see example JWK)

The client protects its private key from unauthorized access, use, and modification.

At runtime, when the bilirubin monitoring service needs to authenticate to the token endpoint, it generates a one-time-use authentication JWT.

JWT Headers:

```
{
  "typ": "JWT",
  "alg": "RS384",
  "kid": "eee9f17a3b598fd86417a980b591fbe6"
}
```

JWT Payload:

```
{
  "iss": "https://bili-monitor.example.com",
  "sub": "bili_monitor",
  "aud": "https://authorize.smarthealthit.org/token",
  "exp": 1422568860,
  "jti": "random-non-reusable-jwt-id-123"
}
```

Using the client's RSA private key, with SHA-384 hashing (as specified for an `RS384` algorithm (`alg`) parameter value in RFC7518), the signed token value is:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzM4NCIsImtpZCI6ImVlZTlmMTdhM2I1OThmZDg2NDE3YTk4MGI1OTFmYm
U2In0.eyJpc3MiOiJiaWxpX21vbml0b3IiLCJzdWIiOiJiaWxpX21vbml0b3IiLCJhdWQiOiJodHRwczovL2F1
dGhvcml6ZS5zbWFydGhlYWx0aGl0Lm9yZy90b2tlbiIsImV4cCI6MTQyMjU2ODg2MCwianRpIjoicmFuZG9tLW
5vbi1yZXVzYWJsZS1qd3QtaWQtMTIzIn0.l2E3-ThahEzJ_gaAK8sosc9uk1uhsISmJfwQOtooEcgUiqkdMFdA
UE7sr8uJN0fTmTP9TUxssFEAQnCOF8QjkMXngEruIL190YVlwukGgv1wazsi_ptI9euWAf2AjOXaPFm6t629vz
dznzVu08EWglG70l41697AXnFK8GUWSBf_8WHrcmFwLD_EpO_BWMoEIGDOOLGjYzOB_eN6abpUo4GCB9gX2-U8
IGXAU8UG-axLb35qY7Mczwq9oxM9Z0_IcC8R8TJJQFQXzazo9YZmqts6qQ4pRlsfKpy9IzyLzyR9KZyKLZalBy
twkr2lW7QU3tC-xPrf43jQFVKr07f9dA
```

Note: to inspect this example JWT, you can visit https://jwt.io. Paste the signed JWT value above into the "Encoded" field, and paste the sample public signing key (starting with the `{"kty": "RSA"` JSON object, and excluding the `{ "keys": [` JWK Set wrapping array) into the "Public Key" box. The plaintext JWT will be displayed in the "Decoded:Payload" field, and a "Signature Verified" message will appear.

For a complete code example demonstrating how to generate this assertion, see: rendered Jupyter Notebook, source .ipynb file.

Requesting an Access Token

A `client_assertion` generated in this fashion can be used to request an access token as part of a SMART App Launch authorization flow, or as part of a SMART Backend Services authorization flow. See complete example:

- SMART App Launch: specification; full example
- SMART Backend Services: specification; full example

6 Symmetric Authentication

- Profile Audience and Scope
- Authentication using a `client_secret`

Profile Audience and Scope

This profile desribes SMART's `client-confidential-symmetric` authentication mechanism. It is intended for for SMART App Launch clients that can maintain a secret but cannot manage asymmetric keypairs. For client that can manage asymmetric keypairs, Asymmetric Authentication is preferred. This profile is not intended for SMART Backend Services clients.

Authentication using a `client_secret`

If a client has registered for Client Password authentication (i.e., it possesses a `client_secret` that is also known to the EHR), the client authenticates by supplying an `Authorization` header with HTTP Basic authentication, where the username is the app's `client_id` and the password is the app's `client_secret`.

Example

If the `client_id` is "my-app" and the `client_secret` is "my-app-secret-123", then the header uses the value B64Encode("my-app:my-app-secret-123"), which converts to `bXktYXBwOm15LWFwcC1zZWNyZXQtMTIz`. This gives the app the Authorization token for "Basic Auth".

GET header:

```
Authorization: Basic bXktYXBwOm15LWFwcC1zZWNyZXQtMTIz
```

7 Token Introspection

SMART on FHIR EHRs SHOULD support Token Introspection, which allows a broader ecosystem of resource servers to leverage authorization decisions managed by a single authorization server. Token Introspection is conducted according to RFC 7662: OAuth 2.0 Token Introspection, with the following additional considerations.

Required fields in the introspection response

In addition to the `active` field required by RFC7662 (a boolean indicating whether the access token is active), the following fields SHALL be included in the introspection response:

- `scope`. As included in the original access token response. The list of scopes granted by the authorization server as a space-separated JSON string.
- `client_id`. As included in the original access token response. The client identifier of the client to which the token was issued.
- `exp`. As included in the original access token response. The integer timestamp indicating when the access token expires.

Conditional fields in the introspection response

In addition to the required fields, the following fields SHALL be included in the introspection response when the specified conditions are met:

- SMART Launch Context. If a launch context parameter defined in Scopes and Launch Context (e.g., `patient` or `intent`) was included in the original access token response, the parameter SHALL be included in the token introspection response.
- ID Token Claims. If an `id_token` was included in the original access token response, the following claims from the ID Token SHALL be included in the Token Introspection response:
  - `iss`
  - `sub`
- ID Token Claims. If an `id_token` was included in the original access token response, the following claims from the ID Token SHOULD be included in the Token Introspection response:
  - `fhirUser`

Authorization to perform Token Introspection

SMART on FHIR EHRs MAY implement access control protecting the Token Introspection endpoint. If access control is implemented, any client authorized to issue Token Introspection API calls SHOULD be able to authenticate to the Token Introspection endpoint using its client credentials. Further considerations for access control are out of scope for the SMART App Launch IG.

Example Request and Response

Example Token Introspection request:

```
POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer 23410913-abewfq.123483

token=2YotnFZFEjr1zCsicMWpAA
```

Example Token Introspection response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
 "active": true,
 "client_id": "07a89bd2-52ce-4576-8b85-71698efa8328",
 "scope": "patient/*.read openid fhirUser",
 "sub": "c91dfe96-731d-4e66-b4f9-01f8f8a4b7b2",
 "patient": "Patient/456",
 "fhirUser": "https://ehr.example.org/fhir/Patient/123",
 "exp": 1597678964,
}
```

8 Conformance

- SMART on FHIR OAuth authorization Endpoints and Capabilities
- FHIR Authorization Endpoint and Capabilities Discovery using a Well-Known Uniform Resource Identifiers (URIs)

The SMART's App Launch specification enables apps to launch and securely interact with EHRs. The specification can be described as a set of capabilities and a given SMART on FHIR server implementation may implement a subset of these. The methods of declaring a server's SMART authorization endpoints and launch capabilities are described in the sections below.

SMART on FHIR OAuth authorization Endpoints and Capabilities

The server SHALL convey the FHIR OAuth authorization endpoints and any *optional* SMART Capabilities it supports using a Well-Known Uniform Resource Identifiers (URIs) JSON file. (In previous versions of SMART, some of these details were also conveyed in a server's CapabilityStatement; this mechanism is now deprecated.)

Capability Sets

A *Capability Set* combines individual capabilities to enable a specific use-case. A SMART on FHIR server SHOULD support one or more *Capability Set*s. Unless otherwise noted, each capability listed is required to satisfy a *Capability Set*. Any individual SMART server will publish a granular list of its capabilities; from this list a client can determine which of these Capability Sets are supported:

External implementation guides MAY define additional capabilities to be discovered through this same mechanism. IGs published by HL7 MAY use simple strings to represent additional capabilities (e.g., `example-new-capability`); IGs published by other organizations SHALL use full URIs to represent additional capabilities (e.g., `http://sdo.example.org/example-new-capability`).

Patient Access for Standalone Apps

1. `launch-standalone`
2. At least one of `client-public` or `client-confidential-symmetric`; and MAY support `client-confidential-asymmetric`
3. `context-standalone-patient`
4. `permission-patient`

Patient Access for EHR Launch (i.e. from Portal)

1. `launch-ehr`
2. At least one of `client-public` or `client-confidential-symmetric`; and MAY support `client-confidential-asymmetric`
3. `context-ehr-patient`
4. `permission-patient`

Clinician Access for Standalone

1. `launch-standalone`
2. At least one of `client-public` or `client-confidential-symmetric`; and MAY support `client-confidential-asymmetric`
3. `permission-user`
4. `permission-patient`

Clinician Access for EHR Launch

1. `launch-ehr`
2. At least one of `client-public` or `client-confidential-symmetric`; and MAY support `client-confidential-asymmetric`
3. `context-ehr-patient` support
4. `context-ehr-encounter` support
5. `permission-user`
6. `permission-patient`

Capabilities

To promote interoperability, the following SMART on FHIR *Capabilities* have been defined. A given set of these capabilities is combined to support a specific use, a *Capability Set*.

Launch Modes

- `launch-ehr`: support for SMART’s EHR Launch mode
- `launch-standalone`: support for SMART’s Standalone Launch mode

Authorization Methods

- `authorize-post`: support for POST-based authorization

Client Types

- `client-public`: support for SMART’s public client profile (no client authentication)

- `client-confidential-symmetric`: support for SMART's symmetric confidential client profile ("client secret" authentication). See [Client Authentication: Symmetric]().
- `client-confidential-asymmetric`: support for SMART's asymmetric confidential client profile ("JWT authentication"). See [Client Authentication: Asymmetric]().

Single Sign-on

- `sso-openid-connect`: support for SMART's OpenID Connect profile

Launch Context

The following capabilities convey that a SMART on FHIR server is capable of providing context to an app at launch time.

Lauch Context for UI Integration

These capabilities only apply during an EHR Launch, and `context-style` only for an embedded EHR Launch.

- `context-banner`: support for "need patient banner" launch context (conveyed via `need_patient_banner` token parameter)
- `context-style`: support for "SMART style URL" launch context (conveyed via `smart_style_url` token parameter). This capability is deemed *experimental*.

Launch Context for EHR Launch

When a SMART on FHIR server supports the launch of an app from *within* an existing user session ("EHR Launch"), the server has an opportunity to pass existing, already-established context (such as the current patient ID) through to the launching app. Using the following capabilities, a server declares its ability to pass context through to an app at launch time:

- `context-ehr-patient`: support for patient-level launch context (requested by `launch/patient` scope, conveyed via `patient` token parameter)
- `context-ehr-encounter`: support for encounter-level launch context (requested by `launch/encounter` scope, conveyed via `encounter` token parameter)

Launch Context for Standalone Launch

When a SMART on FHIR server supports the launch of an app from *outside* an existing user session ("Standalone Launch"), the server may be able to proactively resolve new context to help establish the details required for an app launch. For example, an external app may request that the SMART on FHIR server should work with the end-user to establish a patient context before completing the launch.

- `context-standalone-patient`: support for patient-level launch context (requested by `launch/patient` scope, conveyed via `patient` token parameter)
- `context-standalone-encounter`: support for encounter-level launch context (requested by `launch/encounter` scope, conveyed via `encounter` token parameter)

Permissions

- `permission-offline`: support for refresh tokens (requested by `offline_access` scope)
- `permission-online`: support for refresh tokens (requested by `online_access` scope)
- `permission-patient`: support for patient-level scopes (e.g. `patient/Observation.rs`)
- `permission-user`: support for user-level scopes (e.g. `user/Appointment.rs`)
- `permission-v1`: support for SMARTv1 scope syntax (e.g., `patient/Observation.read`)
- `permission-v2`: support for SMARTv2 granular scope syntax (e.g., `patient/Observation.rs?category=http://terminology.hl7.org/CodeSystem/observation-category|vital-signs`)

FHIR Authorization Endpoint and Capabilities Discovery using a Well-Known Uniform Resource Identifiers (URIs)

The authorization endpoints accepted by a FHIR resource server are exposed as a Well-Known Uniform Resource Identifiers (URIs) (RFC5785) JSON document.

FHIR endpoints requiring authorization SHALL serve a JSON document at the location formed by appending `/.well-known/smart-configuration` to their base URL. Contrary to RFC5785 Appendix B.4, the `.well-known` path component may be appended even if the FHIR endpoint already contains a path component.

Responses for `/.well-known/smart-configuration` requests SHALL be JSON, regardless of `Accept` headers provided in the request.

- clients MAY omit an `Accept` header
- servers MAY ignore any client-supplied `Accept` headers
- servers SHALL respond with `application/json`

Sample Request

Sample requests:

Base URL "fhir.ehr.example.com"

```
GET /.well-known/smart-configuration HTTP/1.1

Host: fhir.ehr.example.com
```

Base URL "www.ehr.example.com/apis/fhir"

```
GET /apis/fhir/.well-known/smart-configuration HTTP/1.1

Host: www.ehr.example.com
```

Response

A JSON document must be returned using the `application/json` mime type.

Metadata

- `issuer`: **CONDITIONAL**, String conveying this system's OpenID Connect Issuer URL. Required if the server's capabilities include `sso-openid-connect`; otherwise, omitted.
- `jwks_uri`: **CONDITIONAL**, String conveying this system's JSON Web Key Set URL. Required if the server's capabilities include `sso-openid-connect`; otherwise, optional.
- `authorization_endpoint`: **REQUIRED**, URL to the OAuth2 authorization endpoint.
- `grant_types_supported`: **REQUIRED**, Array of grant types supported at the token endpoint. The options are "authorization_code" (when SMART App Launch is supported) and "client_credentials" (when SMART Backend Services is supported).
- `token_endpoint`: **REQUIRED**, URL to the OAuth2 token endpoint.
- `token_endpoint_auth_methods_supported`: **OPTIONAL**, array of client authentication methods supported by the token endpoint. The options are "client_secret_post", "client_secret_basic", and "private_key_jwt".
- `registration_endpoint`: **OPTIONAL**, If available, URL to the OAuth2 dynamic registration endpoint for this FHIR server.
- `scopes_supported`: **RECOMMENDED**, Array of scopes a client may request. See scopes and launch context. The server SHALL support all scopes listed here; additional scopes MAY be supported (so clients should not consider this an exhaustive list).
- `response_types_supported`: **RECOMMENDED**, Array of OAuth2 `response_type` values that are supported. Implementers can refer to `response_type`s defined in OAuth 2.0 (RFC 6749) and in OIDC Core.

- `management_endpoint`: **RECOMMENDED**, URL where an end-user can view which applications currently have access to data and can make adjustments to these access rights.
- `introspection_endpoint` : **RECOMMENDED**, URL to a server's introspection endpoint that can be used to validate a token.
- `revocation_endpoint` : **RECOMMENDED**, URL to a server's revoke endpoint that can be used to revoke a token.
- `capabilities`: **REQUIRED**, Array of strings representing SMART capabilities (e.g., `sso-openid-connect` or `launch-standalone`) that the server supports.
- `code_challenge_methods_supported`: **REQUIRED**, Array of PKCE code challenge methods supported. The `S256` method SHALL be included in this list, and the `plain` method SHALL NOT be included in this list.

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json


{
  "issuer": "https://ehr.example.com",
  "jwks_uri": "https://ehr.example.com/.well-known/jwks.json",
  "authorization_endpoint": "https://ehr.example.com/auth/authorize",
  "token_endpoint": "https://ehr.example.com/auth/token",
  "token_endpoint_auth_methods_supported": [
    "client_secret_basic",
    "private_key_jwt"
  ],
  "grant_types_supported": [
    "authorization_code",
    "client_credentials"
  ],
  "registration_endpoint": "https://ehr.example.com/auth/register",
  "scopes_supported": ["openid", "profile", "launch", "launch/patient",
"patient/*.rs", "user/*.rs", "offline_access"],
  "response_types_supported": ["code"],
  "management_endpoint": "https://ehr.example.com/user/manage",
  "introspection_endpoint": "https://ehr.example.com/user/introspect",
  "revocation_endpoint": "https://ehr.example.com/user/revoke",
  "code_challenge_methods_supported": ["S256"],
  "capabilities": [
    "launch-ehr",
    "permission-patient",
    "permission-v2",
    "client-public",
    "client-confidential-symmetric",
    "context-ehr-patient",
    "sso-openid-connect"
  ]
}
```

9 Best Practices

Considerations for Scope Consent (Non-Normative)

In 3rd-party authorization scenarios (where the client and the resource server are not from the same organization), it is a common requirement for authorization servers to obtain the user's consent prior to granting the scopes requested by the client. In order to collect the required consent in a transparent manner, it is important that the authorization server presents a summary of the requested scopes in concise, plain language that the user understands.

The responsibility of supporting transparent consent falls on both the authorization server implementer as well as the client application developer.

Client Application Considerations

- In a complex authorization scenario involving user consent, the complexity of the authorization request presented to the user should be considered and balanced against the concept of least privilege. Make effective use of both wildcard and SMART 2.0 fine grained resource scopes to reduce the number and complexity of scopes requested. The goal is to request an appropriate level of access in a transparent manner that the user fully understands and agrees with.

Authorization Server Considerations

- For each requested scope- present the user with both a short and long description of the access requested. The long description may be available in a pop-up window or some similar display method. These descriptions should be in plain language, localized to the language set in the user's browser.
- Consider publishing consent design documentation for client developers- including user interface screenshots and full scope description metadata. This will provide valuable transparency to client developers as they make decisions on what access to request at authorization time.
- Avoid industry jargon when describing a given scope to the user. For example, an average patient may not know what is meant if a client application is requesting for access to their "Encounters".
- If using the experimental query-based scopes, consider how the query will be represented in plain language. If the query cannot easily be explained in a single sentence, adjustment of the requested scope should be considered or proper documentation provided to educate the intended user population.

App and Server developers should consider trade-offs associated with confidential vs public app architectures

1. Persistent access is important for providing a seamless consumer experience, and Refresh Tokens are the mechanism SMART App Launch defines for enabling persistent access. If an app is ineligible for refresh tokens, the developer is likely to seek other means of achieving

this (e.g., saving a user's password and simulating login; or moving to a cloud-based architecture even though there's no use case for managing data off-device).

2. Client architectures where data pass through or are stored in a secure backend server (e.g., many confidential clients) can offer more secure {refresh token :: client} binding, but are open to certain attacks that purely-on-device apps are not subject to (e.g., cloud server becomes compromised and tokens/secrets leak). A breach in this context can be widespread, across many users.
3. Client architectures where data are managed exclusively on end-user devices (e.g., many public clients including most native apps today, where an app is only registered once with a given EHR) are open to certain attacks that confidential clients can avoid (e.g., a malicious app on your device might steal tokens from a valid app, or might impersonate a valid app). A breach in this context is more likely to be isolated to a given user or device.

The choice of app architecture should be based based on context. Apps that already need to manage data in the cloud should consider a confidential client architecture; apps that don't should consider a purely-on-device architecture. But this decision only works if refresh tokens are available in either case; otherwise, app developers will switch architectures just to be able to maintain persistent access, even if the overall security posture is diminished.

Best Practices

This page reflects best practices established at the time of publication. For up-to-date community discussion, see SMART on FHIR Best Practices on the HL7 Confluence Site

Best practices for server developers include

- Remind users which apps have offline access (keeping in mind that too many reminders lead to alert fatigue)
- Mitigate threats of compromised refreshed tokens
- Expire an app's authorization if a refresh token is used more than once (see OAuth 2.1 section 6.1)
- Consider offering clients a way to bind refresh tokens to asymmetric secrets managed in hardware
- E.g., per-device dynamic client registration (see ongoing work on UDAP specifications)
- E.g., techniques like the draft DPOP specification

Best practices for app developers include

- Ensure that refresh tokens are never used more than once
- Take advantage of techniques to bind refresh tokens to asymmetric secrets managed in hardware, when available (see above)
- If an app only needs to connect to EHR when the user is present, maintain secrets with best-available protection (e.g., biometric unlock)
- Publicly document any code of conduct that an app adheres to (e.g., CARIN Alliance code of conduct)

10 Examples

- SMART App Launch Examples
- SMART Backend Services Examples

SMART App Launch Examples

These examples demonstrate all steps involved in the SMART App Launch authorization process.

- Public client
- Confidential client, asymmetric authentication
- Confidential client, symmetric authentication

SMART Backend Services Examples

This example demonstrates all steps involved in the SMART Backend Services authorization process.

- Backend Services

11 Artifacts Summary

Contents:

This page provides a