FHIR Bulk Data Export API Proposal

Request Flow

Authorization

Bulk data servers should implement the OAuth based SMART backend services authorization process. On the requests outlined below, clients should include an Authorization header containing the bearer token received from the authorization flow. If the server responds to a request with a 401 Unauthorized header, the client should follow the authorization flow to obtain a new token.

Bulk Data Kick-off Request

This FHIR Operation initiates the asynchronous generation of data files for all patients, a group of patients, or all available data contained in a FHIR server.

Note: Only data the client application has authorization to access and that the relevant business agreements allow should be returned.

Endpoint - All Patients

GET [fhir base]/Patient/$export

Endpoint - Group of Patients

GET [fhir base]/Group/[id]/$export

FHIR Operation to obtain data on all patients listed in a single FHIR Group Resource.

Note: How these groups are defined will be implementation specific for each clinical system. For example, a payer may send a healthcare institution a roster file that can be imported into their EHR to create or update a FHIR group. Group membership could be based upon explicit attributes of the patient, such as: age, sex or a particular condition such as PTSD or Chronic Opioid use, or on more complex attributes, such as a recent inpatient discharge or membership in the population used to calculate a quality measure. Although, FHIR based group management is out of scope for the bulk data project, it would be a valuable project.

Endpoint - System Level Export

GET [fhir base]/$export
Export data from a FHIR server whether or not it is associated with a patient. This supports use cases like backing up a server or exporting terminology data by restricting the resources returned using the _type parameter.

Headers

- **Accept (required)**
  
  Specifies the format of the optional OperationOutcome response to the kick-off request. Currently, only application/fhir+json is supported.

- **Prefer (required)**
  
  Specifies whether the response is immediate or asynchronous. Currently must be set to respond-async.

Query Parameters

- **_outputFormat (string, optional, defaults to application/fhir+ndjson)**
  
  The format for the generated bulk data files. Currently, ndjson must be supported, though servers may choose to also support other output formats. Servers should support the full content type of application/fhir+ndjson as well as abbreviated representations including application/ndjson and ndjson.

- **_since (FHIR instant type, optional)**
  
  Resources updated after this period will be included in the response

  Note: This parameter was named start in an earlier version of this proposal

- **_type (string of comma-delimited FHIR resource types, optional)**

  Only resources of the specified resource types(s) will be included in the response. If this parameter is omitted, the server should return all supported resources that the client has authorization to access and that the relevant business agreements allow. The Patient Compartment should act as a point of reference for recommended resources to be returned as well as other resources outside of the patient compartment that are helpful in interpreting the patient data such as Organization and Practitioner.

  Resource references should be relative uris with the format <resource type>/<id>, or absolute uris with the same structure rooted in the base url for the server from which the export was performed. References will be resolved looking for a resource with the specified type and id within the file set.

  Note: Some implementations may limit the resources returned to specific subsets of FHIR like those defined in the Argonaut Implementation Guide

Experimental Query Parameters
As a community, we’ve identified use cases for finer-grained, client-specified filtering. For example, some clients may want to retrieve only active prescriptions (rather than historical prescriptions), or only laboratory observations (rather than all observations). We have considered several approaches to finer-grained filtering, including FHIR’s GraphDefinition, the Clinical Query Language, and FHIR’s REST API search parameters. We expect this will be an area of active exploration, so for the time being we’re defining an experimental syntax based on search parameters that works side-by-side with our coarse-grained _type-based filtering.

To request finer-grained filtering, a client can supply a _typeFilter parameter alongside the _type parameter. The value of the _typeFilter parameter is a comma-separated list of FHIR REST API queries that further restrict the results of the query.

Understanding _typeFilter is optional for servers; clients should be robust to servers that ignore _typeFilter.

**Note for client developers**: Because both _typeFilter and _since can restrict the results returned, the interaction of these parameters may be surprising. Think carefully through the implications when constructing a query with both of these parameters.

### Example Request with _typeFilter

The following is an export request for MedicationRequest resources and Condition resources, where the client would further like to restrict MedicationRequests to requests that are active, or else completed after July 1 2018. This can be accomplished with two subqueries, joined together with a comma for a logical "or":

- MedicationRequest?status=active
- MedicationRequest?status=completed&date=gt2018-07-01T00:00:00Z

To create a _typeFilter parameter, a client should URL encode these two subqueries and join them with `. Newlines and spaces have been added for clarity, and would not be included in a real request:

```plaintext
$export?
  _type=
  MedicationRequest,
  Condition&_typeFilter=
  MedicationRequest%3Fstatus%3Dactive,
  MedicationRequest%3Fstatus%3Dcompleted%26date%3Dgt2018-07-01T00%3A00%3A00Z
```

Note: the Condition resource is included in _type but omitted from _typeFilter because the client intends to request all Condition resources without any filters.

### Response - Success

- HTTP Status Code of 202 Accepted
- Content-Location header with an absolute url for subsequent status requests
- Optionally a FHIR OperationOutcome in the body

### Response - Error (eg. unsupported search parameter)
HTTP Status Code of 4xx or 5xx

The body MUST be a FHIR OperationOutcome in JSON format

If a server wants to prevent a client from beginning a new export before an in-progress export is completed, it should respond with a 429 status and a Retry-After header, following the rate-limiting advice for "Bulk Data Status Request" below.

---

**Bulk Data Delete Request:**

After a bulk data request has been started, clients can send a delete request to the url provided in the Content-Location header to cancel the request.

**Endpoint**

DELETE [polling content location]

**Response - Success**

- HTTP Status Code of 202 Accepted
- Optionally a FHIR OperationOutcome in the body

**Response - Error Status**

- HTTP status code of 4xx or 5xx
- The body MUST be a FHIR OperationOutcome in JSON format

---

**Bulk Data Status Request:**

After a bulk data request has been started, clients can poll the url provided in the Content-Location header, using any access token valid at the initial $export endpoint, to obtain the status of the request.

Note: Clients should follow an exponential backoff approach when polling for status. Servers may supply a Retry-After header with a http date or a delay time in seconds. When provided, clients should use this information to inform the timing of future polling requests. If a client is polling too frequently, the server should respond with a 429 status code in addition to a Retry-After header, and optionally an OperationOutcome with further explanation.

Note: The Accept header for this request should be application/json. In the case that errors prevent the export from completing, the response will contain a JSON-encoded FHIR OperationOutcome resource.
Endpoint

GET [polling content location]

Response - In-Progress Status

- HTTP Status Code of 202 Accepted
- Optionally an \texttt{x-progress} header with a text description of the status of the request that’s less than 100 characters. The format of this description is at the server’s discretion and may be a percentage complete value or a more general status such as "in progress". Clients can try to parse this value, display it to the user, or log it.

Response - Error Status

- HTTP status code of 5xx
- The body MUST be a FHIR OperationOutcome in JSON format
- Even if some resources cannot successfully be exported, the overall export operation may still succeed. In this case, the \texttt{Response.error} array of the completion \texttt{Response} must be populated (see below) with one or more files in ndjson format containing \texttt{OperationOutcome} resources to indicate what went wrong.

Response - Complete Status

- HTTP status of 200 OK
- Content-Type header of \texttt{application/json}
- Optionally an \texttt{Expires} header indicating when the files listed will no longer be available.

- A body containing a json object providing metadata and links to the generated bulk data files.

Required Fields:

- \texttt{transactionTime} - a FHIR instant type that indicates the server’s time when the query is run. No resources that have a modified data after this instant should be in the response.
- \texttt{request} - the full url of the original bulk data kick-off request
- \texttt{requiresAccessToken} - boolean value indicating whether downloading the generated files will require an authentication token. Note: This may be false in the case of signed S3 urls or an internal file server within an organization’s firewall.
- \texttt{output} - array of bulk data file items with one entry for each generated file. Note: If no data is returned from the kick-off request, the server should return an empty array.
error - array of error file items following the same structure as the output array. Note: If no errors occurred, the server should return an empty array. Note: Only the OperationOutcome resource type is currently supported, so a server will generate ndjson files where each row is an OperationOutcome resource.

Each file item should contain the following fields:

- type - the FHIR resource type that is contained in the file. Note: Each file may only contain resources of one type, but a server may create more than one file for each resource type returned. The number of resources contained in a file is may vary between servers. If no data is found for a resource, the server should not return an output item for it in the response.
- url - the path to the file. The format of the file should reflect that requested in the _outputFormat parameter of the initial kick-off request.

Each file item may optionally contain the following field:

- count - the number of resources in the file, represented as a JSON number.

Example response body:

```
{
  "transactionTime": "[instant]",
  "request": "[base]/Patient/$export?_type=Patient,Observation",
  "requiresAccessToken": true,
  "output": [{
    "type": "Patient",
    "url": "http://serverpath2/patient_file_1.ndjson"
  },{
    "type": "Patient",
    "url": "http://serverpath2/patient_file_2.ndjson"
  },{
    "type": "Observation",
    "url": "http://serverpath2/observation_file_1.ndjson"
  }],
  "error": [{
    "type": "OperationOutcome",
    "url": "http://serverpath2/err_file_1.ndjson"
  }]
}
```

File Requests:

Using the urls supplied in the completed status request body, clients can download the generated bulk data files (one or more per resource type). Note: These files may be
served by a file server rather than a FHIR specific server. Also, if the requiresAccessToken field in the status body is set to true the request must include a valid access token in the Authorization header in these requests (i.e., Authorization: Bearer {{token}}).

**Endpoint**

GET [url from status request output field]

**Headers**

- Accept (optional, defaults to application/fhir+ndjson)

Specifies the format of the file being returned. Optional, but currently only application/fhir+ndjson is supported.

**Response - Success**

- HTTP status of 200 OK
- Content-Type header of application/fhir+ndjson
- Body of FHIR resources in newline delimited json - ndjson format

**Response - Error**

- HTTP Status Code of 4xx or 5xx

---

**SMART Backend Services: Authorization Guide**

**Profile audience and scope**

This profile is intended to be used by developers of back-end services that need to access FHIR resources by requesting access tokens from OAuth 2.0 compliant authorization servers. This profile assumes that a backend service has been authorized up-front, at registration time, and describes the runtime process by which the service acquires an access token that can be used to communicate with a FHIR Resource Server.

Use this profile when the following conditions apply:
• The service runs automatically, without user interaction
• The service is able to protect a private key

Examples

• An analytics platform or data warehouse that periodically performs a bulk data export from an electronic health record system to provide insights into 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.

• 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.

Registering a SMART Backend Service (communicating public keys)

Before a SMART backend service can run against an EHR, the service 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.

No matter how a backend service registers with an EHR's authorization service, a backend service should communicate its public key to the SMART EHR using a JSON Web Key Set (JWKS) by one of the following techniques:

1. JWKS URL (preferred). This URL communicates the TLS-protected endpoint where the service's public JSON Web Keys can be found. When provided, this URL will match the jku header parameter in the service's Authorization JWTs. An advantage of this approach is that it allows a client to rotate its own keys by updating the hosted content at the JWKS URL.

2. JWKS directly (allowed, not preferred). If a backend service cannot host a JWKS at a TLS-protected URL, it may supply a JWKS directly to the EHR at registration time. A limitation of this approach is that it does not enable the client to rotate its keys in-band.

It is recommended that EHRs should be capable of validating signatures using RS384 and ES384; and that backend services be capable of generating signatures using one of these two algorithms. Over time, we expect recommended algorithms to
evolve, so while this specification recommends algorithms for interoperability, it does not mandate any algorithm.

No matter how a JWKS is communicated to the EHR, each key in the JWKS must be an asymmetric key that includes kty and kid properties, and whose content is conveyed using "bare key" properties (i.e., direct base64 encoding of key material as integer values). This means that:

- For RSA public keys, each MUST include \( n \) and \( e \) values (modulus and exponent)
- For ECDSA public keys, each MUST include \( \text{crv} \), \( x \), and \( y \) values (curve, x-coordinate, and y-coordinate, for EC keys)

Upon registration, the server assigns a client_id, which the client uses when obtaining an access token.

**Obtaining an access token**

By the time a backend service has been registered with the EHR, the key elements of organizational trust are already established. That is, the app is considered "pre-authorized" to access clinical data. Then, at runtime, the backend service must obtain an access token in order to work with clinical data. Such access tokens can be issued automatically, without need for human intervention, and they are short-lived, with a recommended expiration time of five minutes.

To obtain an access token, the service uses an OAuth 2.0 client credentials flow, with a JWT assertion as its client authentication mechanism. The exchange, depicted below, allows the backend service to authenticate to the EHR and request a short-lived access token:
Protocol details

Before a backend service can request an access token, it must generate a one-time-use JSON Web Token (JWT) that will be used to authenticate the service to the EHR's authorization server. The authentication JWT is constructed with the following claims, and then signed with the backend service's private RSA key (RSA SHA-384 signature). For a practical reference on JWT, as well as debugging tools and client libraries, see https://jwt.io.
Authentication JWT Header Values

<table>
<thead>
<tr>
<th>Key</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alg</td>
<td>required</td>
<td>The algorithm used for signing the authentication JWT (e.g., <code>RS384</code>, <code>EC384</code>).</td>
</tr>
<tr>
<td>kid</td>
<td>required</td>
<td>The identifier of the key-pair used to sign this JWT. This identifier MUST be unique within the backend services's JWK Set.</td>
</tr>
<tr>
<td>typ</td>
<td>required</td>
<td>Fixed value: <code>JWT</code>.</td>
</tr>
<tr>
<td>jku</td>
<td>optional</td>
<td>The URL to the JWK Set containing the public key(s). When present, this should match a value that the backend service supplied to the EHR at client registration time.</td>
</tr>
</tbody>
</table>

Authentication JWT Claims

<table>
<thead>
<tr>
<th>Key</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iss</td>
<td>required</td>
<td>The service's <code>client_id</code>, as determined during registration with the EHR's authorization server (note that this is the same as the value for the <code>sub</code> claim)</td>
</tr>
<tr>
<td>sub</td>
<td>required</td>
<td>The service's <code>client_id</code>, as determined during registration with the EHR's authorization server (note that this is the same as the value for the <code>iss</code> claim)</td>
</tr>
<tr>
<td>aud</td>
<td>required</td>
<td>The EHR authorization server's &quot;token URL&quot; (the same URL to which this authentication JWT will be posted -- see below)</td>
</tr>
<tr>
<td>exp</td>
<td>required</td>
<td>Expiration time integer for this authentication JWT, expressed in seconds since the &quot;Epoch&quot; (1970-01-01T00:00:00Z UTC). This time MUST be no more than five minutes in the future.</td>
</tr>
<tr>
<td>jti</td>
<td>required</td>
<td>A nonce string value that uniquely identifies this authentication JWT.</td>
</tr>
</tbody>
</table>

After generating an authentication JWT, the service requests a new access token via HTTP POST to the EHR authorization server's token endpoint URL, using content-type `application/x-www-form-urlencoded` with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required/Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope</td>
<td>required</td>
<td>The scope of access requested. See note about scopes below</td>
</tr>
<tr>
<td>grant_type</td>
<td>required</td>
<td>Fixed value: <code>client_credentials</code></td>
</tr>
<tr>
<td>client_assertion_type</td>
<td>required</td>
<td>Fixed value: <code>urn:ietf:params:oauth:client-assertion-type:jwt-bearer</code></td>
</tr>
</tbody>
</table>
Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_assertion</td>
<td>required</td>
<td>Signed authentication JWT value (see above)</td>
</tr>
</tbody>
</table>

The access token response is a JSON object, with the following properties:

**Access token response: property names**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access_token</td>
<td>required</td>
<td>The access token issued by the authorization server.</td>
</tr>
<tr>
<td>token_type</td>
<td>required</td>
<td>Fixed value: bearer.</td>
</tr>
<tr>
<td>expires_in</td>
<td>required</td>
<td>The lifetime in seconds of the access token. The recommended value is 300, for a five-minute token lifetime.</td>
</tr>
<tr>
<td>scope</td>
<td>required</td>
<td>Scope of access authorized. Note that this can be different from the scopes requested by the app.</td>
</tr>
</tbody>
</table>

**Server Obligations for Signature Verification**

Servers SHALL follow all requirements defined in [Section 3 of RFC7523](https://tools.ietf.org/html/rfc7523).

In addition, we require that servers SHALL:

- validate the signature on the JWT
- check that the JWT `exp` claim is valid
- check that the JWT `aud` claim matches the server's OAuth token URL (the URL to which the token was POSTed)
- check that this is not a `jti` value previously encountered for the given `iss` within the maximum allowed authentication JWT lifetime (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 server follows this algorithm:

1. If the `jku` header is present, verify that the `jku` is whitelisted (i.e., that it matches the value supplied at registration time for the specified `client_id`).
   - If the `jku` header is not whitelisted, the signature verification fails.
If the jku header is whitelisted, create a set of potential keys by dereferencing the jku URL. Proceed to step 3.

2. If jku is absent, create a set of potential key sources consisting of: all keys found by dereferencing the registration-time JWKS URI (if any) + any keys supplied in the registration-time JWKS (if any). Proceed to step 3.

3. Filter the potential keys to retain only those where the kid matches the value supplied in the client’s JWK header, and the kty is consistent with the signature algorithm used for the JWT (e.g., RSA for a JWT using an RSA-based signature, or EC for a JWT using an EC-based signature).

4. Attempt to verify the JWK using each key in the potential keys list.
   i. If any attempt succeeds, the signature verification succeeds.
   ii. If all attempts fail, the signature verification fails.

If an error is encountered during the authorization process, servers SHALL respond with errors as defined by the OAuth 2 specification. Servers SHOULD also include an error_uri and error_description as defined by OAuth 2.

**Scopes**

As there is no user or launch context when performing backed services authorization, the existing SMART on FHIR scopes are not appropriate. Instead, applications use system scopes, which have the format system/(:resourceType|*).(read|write|*). These have the same meanings as their matching user/(:resourceType|*).(read|write|*) scopes, but associated with the permissions of the authorized client instead of a human end-user.

**Worked example**

Assume that a "bilirubin result monitoring" service has registered with the EHR’s authorization server, establishing the following

- JWT "issuer" URL: bili_monitor
- OAuth2 client_id: bili_monitor
- RSA public key

Separately, the service also maintains its RSA private key.

To obtain an access token at runtime, the bilirubin monitoring service wants to start monitoring some bilirubin values. It needs to obtain an OAuth2 token with the scopes system/*.read system/CommunicationRequest.write. To accomplish this, the service must first generate a one-time-use authentication JWT with the following claims:

1. Generate a JWT to use for client authentication:
2. Generate an RSA SHA-384 signed JWT over these claims

Using the service's RSA private key, the signed token value is:

```json
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtldXNpb25faWQiOiJiaWxpIiwiY2Fya2VsIjoxfQ.
eyJpc3MiOiJiaWxpIiwiZ2VuZXJhdG9yX3N0YW5kJnVpbmc6aW5nIiwiaXNzIjoiYXN0Zml0IiwiZmFsc2U6
Y2VydGlzaWQ6aW5nIiwidHlwZiOiJleHBsb3JhbHMiLCJyZXNpemUiOiJiaWxpIiwiY2Fya2VsIjoxfQ.
eyJpc3MiOiJiaWxpIiwiY2Fya2VsIjoxfQ.
```

(Note: to inspect this example JWT, you can visit [https://jwt.io](https://jwt.io), choose RS384, paste in the provided RSA keys, and then paste the JWT value into the "encoded" field.)

3. Obtain an access token

The service then calls the SMART EHR’s "token endpoint" using the one-time use authentication JWT as its client authentication mechanism:

**Request**

```
POST /token HTTP/1.1
Host: authorize.smarthealthit.org
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&scope=system%2F*.read%20system%2FCommunicationRequest&
write&client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer
client_assertion=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtldXNpb25faWQiOiJiaWxpIiwiY2Fya2VsIjoxfQ.
```

**Response**

```json
{
  "access_token": "m7rt6i7s9nuxkjvi8v8x",
  "token_type": "bearer",
  "expires_in": 300,
  "jti": "random-non-reusable-jwt-id-123"
}
```
Out of scope in v1 of this specification

- Legal framework for sharing data between partners - BAAs, SLAs, DUAs should continue to be negotiated out-of-band
- Real-time data (although data loaded through bulk data can be supplemented at with synchronous FHIR REST API calls)
- Data transformation and transmission - different step of the ETL process
- Patient matching (although, it’s possible to include identifiers like subscriber number in FHIR resources)
- Management of FHIR groups within the clinical system - the bulk data operation will require a valid group id, but does not specify how FHIR Groups resources are created and maintained within a system