The Subspace Communication Framework

Last Reviewed: 7/13/2020

Introduction 3

Design Overview 3

The FHIRcast Model 3

Initialization Sequence 4

1. Session Discovery 4

2. OAuth Authorization 7

3. The Subscription Handshake 11

Message Passing 15

To Hyperdrive 15

From Hyperdrive 17

Introduction

The Subspace Communication Framework (Subspace, for short) serves as the Hyperdrive replacement for the COM-based integration pattern that EDManager provided for the native Windows Hyperspace application. This document outlines the steps that an external application must take to initiate and communicate via a Subspace integration with Hyperdrive.

This document does not provide details for how any integration functions once it is initiated, beyond a general format of communication between Hyperdrive and the external application.

Design Overview

Subspace is a pattern for asynchronous, asymmetric, semi-event-driven integrations between Hyperdrive and external applications. The pattern follows these general steps:

1. An external application discovers the URL for a local web server at which Hyperdrive is listening and a unique identifier for Hyperdrive.
2. The application obtains an access token associated with permissions to send and/or receive certain events to and/or from Hyperdrive.
3. Using the information obtained in the previous two steps, the application initiates a subscription handshake over HTTPS and Secure WebSockets (WSS) with Hyperdrive's Subspace server.
4. After the handshake is complete, Hyperdrive and the application pass messages over HTTPS (messages to Hyperdrive) and WSS (messages from Hyperdrive).

The FHIRcast Model

Subspace is based off of the Feb 2020 Ballot FHIRcast Spec.

We recommend that any developer wishing to integrate with Hyperdrive via Subspace familiarize themselves with FHIRcast.

In FHIRcast, there is a single "hub" (Hyperdrive) and any number of "subscribers" (external applications). Subscribers send subscription requests to the hub for events for which the hub dispatches notifications during relevant workflows. When the hub dispatches a notification for an event, all subscribers subscribed to that event will receive the notification.

Subscribers may also communicate to the hub directly through requests to change context. In response to these requests, the hub may or may not dispatch one or more event notifications that will go out to all appropriate subscribers, informing them of the change in context.

As an implementation of FHIRcast, Subspace supports all the events in the FHIRcast event catalog. However, not all the events that Subspace supports are part of FHIRcast. For Epic-specific events, please refer to the relevant Epic-provided specs for the event's integration details and how they might differ from FHIRcast's specifications.

Initialization Sequence

1. Session Discovery

To communicate with Hyperdrive via Subspace, an external application must discover the following pieces of information:

- **Hub URL** – the URL at which Hyperdrive's Subspace server is listening
- **Hub Topic** – a unique identifier for this instance of Hyperdrive

Any application can do this by running the **Hyperdrive Launcher**, although some integrations provide other mechanisms for session discovery. Those other mechanisms, if they exist for your integration, are covered in your integration's spec document.

Hyperdrive Launcher

The Hyperdrive Launcher is an executable named "**Launcher.exe**" that is installed with Hyperdrive in "**C:\Program Files (x86)\Epic\Hyperdrive\Hyperspace**".

The Hyperdrive Launcher can find existing instances of Hyperdrive that are running on the user's workstation or launch new ones connected to a given environment. It is designed specifically to be called by external applications to facilitate a Subspace integration.

The Launcher.exe executable must run from "C:\Program Files (x86)\Epic\Hyperdrive\Hyperspace" or it will not function correctly.

Prerequisites

To use the Hyperdrive Launcher, you must know the **Customer ID** and the **Environment Name** for an Epic environment to which you would like Hyperdrive to be connected.

Customer Ids and Environment Names are derived from the config files in "**C:\Program Files (x86)\Epic\Hyperdrive\Config**". Each config file corresponds to a single Customer ID which is the prefix for the file. For example, 861Config.json corresponds to the Customer ID 861.

Each config file contains any number of environments defined under the "Environments" node. The Environment Name of an environment is the name of the node representing the environment. For example, consider the following contents of 861Config.json:

```
{
    "CustomerName": "Epic Medical Center",
    "Environments": {
        "PROD": {
                    "DisplayName": "Production Environment",
                    "HSWebServerURL": "https://epic.example.com/PRODUCTION"
        }
        "SUPPORT": {
                    "DisplayName": "Support Environment",
                    "HSWebServerURL": "https://epic.example.com/SUPPORT"
        }
    }
}
```

The names for the environments for this file would be “PROD” and “SUPPORT” (not “Production Environment” and “Support Environment”).

Input

Arguments are passed to the Launcher.exe executable via the command line. Each argument must be of the form “<KEY>=<VALUE>”

Key	Optionality	Value
id	Required	a valid Customer ID
env	Required	a valid Environment Name
tz	Optional	a valid time zone identifier*
mode	Optional	“Auto”, “Find”, or “Launch” (defaults to “Auto”)

* Contact your Epic representative for a list of valid time zone identifiers for the environment to which you are trying to connect, as these vary by environment.

Behavior

If “mode=Auto” is passed or the “mode” parameter is not passed at all, the Hyperdrive Launcher (referred to as “launcher” below) will first check to see if there are any running Hyperdrive instances connected to the environment with the passed Customer ID and Environment Name (which uniquely identify an environment).

If the launcher discovers such a Hyperdrive instance, it will communicate the details of the instance back to the caller of the executable via standard out (see the Output section below).

If the launcher does not discover an appropriate Hyperdrive instance, it will attempt to launch a new Hyperdrive instance connected to the environment with the passed Customer Id and Environment. Once the launched Hyperdrive instance has successfully connected to its environment, the launcher will communicate the details of the instance back to the caller of the executable via standard out (see the Output section below).

If the “tz” parameter is passed to the launcher, the launcher will launch or find a Hyperdrive that is both connected to the appropriate environment AND operating in the indicated time zone.

If “mode=Find” is passed to the launcher, the launcher will only look for existing Hyperdrive instances connected to the appropriate environment and time zone (if applicable) and return the details of the first instance it finds.

If “mode=Launch” is passed to the launcher, the launcher skips looking for existing Hyperdrive instances and will always launch a new Hyperdrive.

Output

The Hyperdrive Launcher communicates results via exit codes and standard out.

Here is a list of the possible exit codes. Any exit code other than 0 is considered a failure.

Code	Meaning

0	Success
1	The Hyperdrive Launcher encountered an unknown error.
2	Invalid arguments passed to the Hyperdrive Launcher.
3	The Hyperdrive Launcher launched a new Hyperdrive instance but timed out while waiting for communication.
4	The Hyperdrive Launcher attempted to launch a new Hyperdrive instance but encountered a process creation failure.
5	The Hyperdrive Launcher launched a new Hyperdrive instance but the new instance was unable to communicate back to the launcher correctly.
6	The Hyperdrive Launcher attempted to launch a new Hyperdrive instance but the passed Customer ID and Environment could not be resolved to a valid, available environment. Or, if a time zone identifier was passed, the environment found could not support the indicated time zone.
7	The Hyperdrive Launcher attempted to launch a new Hyperdrive instance but the new instance could not connect to the specified environment.
8	The Hyperdrive Launcher attempted to launch a new Hyperdrive instance but the new instance could not correctly set up its Subspace web server.

If the exit code for the Hyperdrive Launcher is 0, it will also write an object to standard out in JSON as follows:

```
{
    "url": "https://example.com/Adfj934-2303",
    "topic": "873njwe98-oo-0",
    "environment": "exampleenv",
    "customerId": "123",
    "timezone": "Pacific",
    "isNew": true
}
```

If "findOnly" is passed to the Hyperdrive Launcher and no running Hyperdrive instance was found, the object will be empty.

A non-empty object representing a Hyperdrive instance contains the following properties:

Property Name	Value
url	Hub URL
topic	Hub Topic

environment	Environment Name of the environment to which Hyperdrive is connected; should match the value of the “env” parameter passed to the Hyperdrive Launcher
customerId	Customer ID of the environment to which Hyperdrive is connected; should match the value of the “id” parameter passed to the Hyperdrive Launcher
timezone	Time zone identifier for the time zone in which the Hyperdrive instance’s is operating; should match the value of the “tz” parameter passed to the Hyperdrive Launcher, if one was passed
isNew	true, if the current invocation of the Hyperdrive Launcher started this instance of Hyperdrive

2. OAuth Authorization

Subspace enforces access through OAuth 2.0 authorization. As such, to initialize a Subspace integration, an external application must use a valid OAuth 2.0 access token associated with one or more scopes corresponding to Subspace events for which they have registered.

An access token may either be retrieved directly through Subspace as described below (see Retrieving an Access Token) or received as part of another mechanism specific to your integration. If going through Subspace directly, you will need to perform a one-time dynamic registration of a client for an install of your application (see Dynamic Client Registration). Other mechanisms for retrieving an access token, if they exist, are covered in your integration’s spec document rather than in this document.

Registering for Subspace Scopes

To receive an OAuth 2.0 access token, an organization must first register for a static Client ID and one or more Subspace scopes with Epic. The organization must also have a public key on file with which to validate access token requests.

This document does not cover this registration process. Please contact your Epic representative(s) for assistance setting up a Client ID for Subspace integration.

Retrieving an Access Token

To retrieve an access token directly from Hyperdrive, an application must first discover Hyperdrive’s Hub URL (see Session Discovery section).

The application then sends an HTTP POST request to <Hub URL>/getaccess. The POST request’s body should be a single Javascript Web Token (JWT) with the following claims:

Identifier	Value
iss	the application’s Client ID (either static or dynamic)
sub	the application’s Client ID (either static or dynamic)
aud	a URI for an Epic OAuth authorization endpoint

exp	a UTC timestamp for when the returned access token should expire
jti	a nonce for protecting against replay attacks

The JWT should be signed with the cryptographic private key corresponding to the public key associated with the application's Client ID.

If the request and JWT are valid and Hyperdrive is able to validate the signature of the JWT using the public key on file for the Client ID in the JWT, the Subspace server will respond with a 200 OK response whose body contains a JSON object with the following properties:

Property Name	Value
access_token	a valid access token to be used with Subspace integration
token_type	"bearer"
expires_in	the number of seconds for which the access token remains valid
scope	**If the Client ID used was dynamic** a comma-delimited list of scope strings that correspond to permissions for Subspace events each scope string takes the form <prefix>/<event-name>.<read/write> where • <prefix> is either "fhircast" (for FHIRcast standard events) or "epic" (for Epic custom events) • <event-name> is the unique string used to identify that event • <read/write> indicates whether this scope corresponds to receive ("read") or send ("write") the event **If the Client ID used was static** "system/DynamicClient.register" which indicates this token can only be used to dynamically register a client (see Dynamic Client Registration)

Example:

```
{
    "access_token": "MTQ0NjJkZmQ5OTM2NDE1ZTZjNGZmZjI3",
    "token_type": "bearer",
    "expires_in": 3600,
    "scope": "fhircast/Patient-open.read,fhircast/Patient-open.write,epic/com.epic.shut
             down.read"
}
```

If the request or JWT was invalid or Hyperdrive could not validate the signature on the JWT, the Subspace server will respond with a 4XX response whose body will contain a JSON object with following properties:

Property Name	Value

error	“invalid_request”, ”invalid_client”, “invalid_grant”, “invalid_scope”, or “unauthorized_client” depending on the type of error
error_description	a more specific error message

Example:

```
{
	"error": "invalid_client",
	"error_description": "Client id not recognized."
}
```

Dynamic Client Registration

The first time an install of an application wishes to integrate with Subspace and begin retrieving access tokens directly, it must first dynamically register itself as a client. To dynamically register a client, an application must have

1. Registered a static Client ID (otherwise known as a Software ID) (see Registering for Subspace Scopes)
2. Used its static Client ID to retrieve an access token that can be used to dynamically register a client (see Retrieving an Access Token)

The application then sends an HTTP POST request to <Hub URL>/register.

The POST must have the following headers:

Header Name	Value
Content-Type	“application/json”
Authorization	“Bearer <ACCESS-TOKEN>” where <ACCESS-TOKEN> is the access token that can be used to dynamically register a client (mentioned above)

The POST request’s body must be a JSON object with the following properties:

Property Name	Value
jwks	a JSON Web Key Set (JWKS) containing an RSA public key to be associated with a new dynamic client
software_id	the application’s static Client ID (otherwise known as a Software ID)

Example:

```
	POST https://subspace.example.com
	Authorization: Bearer i8hweunweunweofiwweoijewiwe
	Content-Type: application/json

	{
		"jwks": {
			"keys": [
				{
```

```
                                "e": "AQAB",
                                "n":"kWp2zRA23Z3vTL4uoe8kTFptxBVFunIoP4t_8TDYJrOb7D1iZNDXVeEsYKp6ppm
                                     rTZDAg-cNOTKLd4M39WJc5FN0maTAVKJc7NxklDeKc4dMe1BGvTZNG4MpWBo-
                                     taKULlYUu0ltYJuLzOjIrTHfarucrGoRWqM0sl3z2-fv9k",
                                "kty": "RSA",
                                "kid": "256"
                         }
                  ]
           },
           "software_id": "M39WJc5FN0maTAVKJc7NxklDeKIrTHfaruc"
    }
```

If the request is valid, the access token has the appropriate permissions, and Hyperdrive recognized the static Client ID, the Subspace server will respond with a 200 OK response whose body contains a JSON object with the following property:

Property Name	Value
client_id	the Client ID of a newly associated dynamic client whose public key is the one provided in the dynamic client registration request and whose Subspace permissions match those registered with the original static client.

Example:

```
{
    "client_id": "MTQ0NjJkZmQ5OTM2NDE1ZTZjNGZmZjI3"
}
```

If the request was invalid, Hyperdrive did not recognize the static Client ID, or the static Client ID was not authorized for dynamic client registration, the Subspace server will respond with a 4XX response whose body will contain a JSON object with following properties:

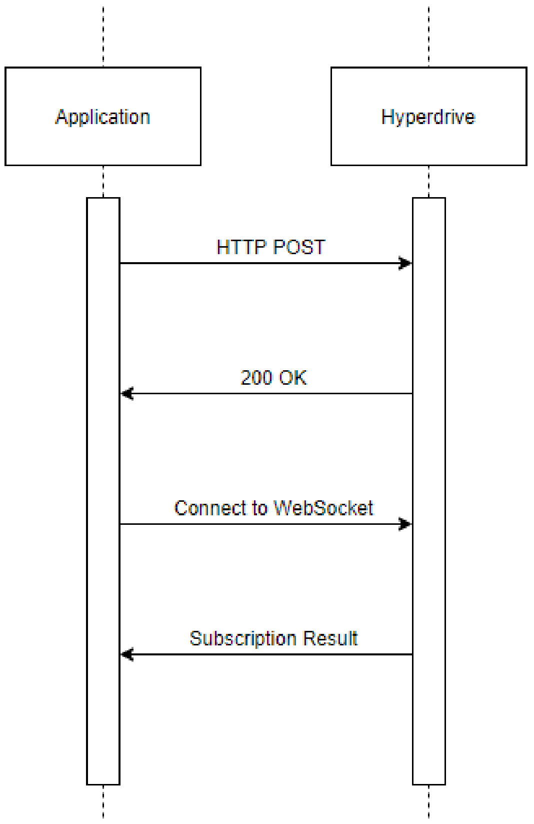
Property Name	Value
error	“invalid_request”, ”invalid_client”, “invalid_scope”, “unauthorized_client” or another error code string depending on the type of error
error_description	a more specific error message

Example:

```
{
    "error": "invalid_scope",
    "error_description": "Access token cannot be used for dynamic client registration."
}
```

3. The Subscription Handshake

To initiate a Subspace integration with Hyperdrive, an external application must successfully complete a Subscription Handshake, pictured below:



Prerequisites

To be able to initiate and successfully complete Subspace's Subscription Handshake, an external application must have the following pieces of information:

- **Hub URL** – the URL at which Hyperdrive's Subspace server is listening
- **Hub Topic** – a unique identifier for this instance of Hyperdrive
- **Access Token** – an OAuth 2.0 access token linked to permissions for Subspace events

For more information on how to obtain Hyperdrive's Hub URL and Hub Topic, refer to the Session Discovery section above.

For more information on how to obtain an Access Token, refer to the OAuth Authorization section above.

Basic Handshake

Sending HTTP Request

The application sends an HTTP Post request to Hyperdrive's Hub URL.

The POST must have the following headers:

Header Name	Value
Content-Type	"application/x-www-form-urlencoded"
Authorization	"Bearer <ACCESS-TOKEN>" where <ACCESS-TOKEN> is the OAuth 2.0 access token mentioned in the OAuth Authorization section.

The POST form body can also have the following key-value pairs:

Key	Value
hub.topic	Hyperdrive's Hub Topic
hub.mode	"subscribe"
hub.events	a comma-delimited listed of event names to which to subscribe
hub.channel.type	"websocket" Subspace does not support the FHIRcast optional value "webhook" for the "hub.channel.type" key-value pair. If "webhook" is sent instead of "websocket," the handshake will fail.

Subspace does not support the optional FHIRcast key "hub.lease_seconds." If included, the value will be ignored.

Example Request:

```
POST https://subspace.example.com
Authorization: Bearer i8hweunweunweofiwweoijewiwe
Content-Type: application/x-www-form-urlencoded

hub.topic=fdb2f928-5546-4f52-87a0-0648e9ded065&hub.mode=subscribe&hub.events=patien
t-open,com.epic.shutdown&hub.channel.type=websocket
```

Receiving HTTP Response

If the application's HTTP request is missing the Authorization header, the Subspace server will respond with 401 Unauthorized and the handshake will fail.

If the application's HTTP request body is missing any values or one of the values is invalid, the Subspace server will respond with 400 Bad Request and the handshake will fail.

If the application's HTTP request body contains a "hub.topic" value that does not match the Hub Topic for Hyperdrive, the Subspace server will respond with 400 Bad Request and the handshake will fail.

Otherwise, the Subspace server will respond with 202 Accepted and the body of the response will contain a string for a WSS endpoint.

Example Accepted Response

```
HTTP/1.1 202 Accepted
```

```
wss://subspace.example.com/ee30d3b9-1558-464f-a299-cbad6f8135de
```

Connecting to WebSocket

If the Subspace server responded with 202 Accepted to the application's HTTP request, the application creates a WSS connection with the Subspace server by connecting to the endpoint received in the Subspace server's response body.

If the application fails to connect or never connects to the WebSocket, the handshake fails.

Receiving WSS Result

Once the Subspace server receives the application's WSS connection, it will send a serialized JSON object representing the final result of the application's subscription request. The object will have the following properties:

Property Name	Value
hub.mode	either "subscribe" or "denied"
hub.topic	the Hyperdrive instance's Hub Topic
hub.events	a comma-delimited list of events to which the application attempted to subscribe
hub.lease_seconds	the number of seconds for which the subscription is valid (only present if hub.mode is "subscribe")
hub.reason	a string message explaining why the subscription was denied (only present if hub.mode is "denied")

If the "hub.mode" property is "subscribe", the handshake has completed successfully and the application can begin sending messages to and receiving messages from Subspace (see the Message Passing section).

If the "hub.mode" property is "denied", the handshake fails. A subscription may be denied at this stage for the following reasons:

- The access token passed in the original request was not recognized by Hyperdrive
- The access token passed in the original request was expired
- The access token passed in the original request was not associated with appropriate scopes for one or more of the asked-for events.

Example Success Response

```
{
    "hub.mode": "subscribe",
    "hub.topic": "fdb2f928-5546-4f52-87a0-0648e9ded065",
    "hub.events": "patient-open,com.epic.shutdown",
    "hub.lease-seconds": 7200
}
```

Example Failure Response

```
{
    "hub.mode": "denied",
    "hub.topic": "fba7b1e2-53e9-40aa-883a-2af57ab4e2c",
    "hub.events": "patient-open,patient-close,com.epic.shutdown",
    "hub.reason": "Subscriber does not have valid authorization for one or more
                   requested events."
}
```

Variations

Re-Subscription

If an application wishes to changes their subscription to Subspace events, they may do so by initiating another Subscription Handshake with the Subspace server. In order to reuse an existing endpoint, the application must add a “hub.channel.endpoint” key-value pair to its initial HTTP request body. The value for this parameter specifies the endpoint to be reused.

The rest of the handshake is the same, with the WSS result being sent through WebSocket as soon as the result can be determined, since the application has already connected in a previous subscription.

If the endpoint passed with the HTTP request is not an endpoint associated with an existing, unexpired subscription, the re-subscription will be denied.

Sending a valid re-subscription **replaces** any original subscription associated with the access token and endpoint pairing used. If re-subscription is denied for any reason, the original subscription will remain unaffected.

Example re-subscription request:

```
POST https://subspace.example.com
Authorization: Bearer i8hweunweunweofiwweoijewiwe
Content-Type: application/x-www-form-urlencoded

hub.topic=fdb2f928-5546-4f52-87a0-0648e9ded065&hub.mode=subscribe&hub.events=patien
t-open,patient-close,com.epic.shutdown&hub.channel.type=webhook&hub.channel.endpoin
t=wss%3A%2F%2Fsubspace.example.com%2Fsession%2Fws%2Fv7tfwuk17a
```

Un-Subscription

If an application wishes to unsubscribe from all events associated with their access token and endpoint pairing, they may do so by initiating another Subscription Handshake with Subspace server. However, in the initial request the value of the “hub.mode” key-value pair must be “unsubscribe” instead of “subscribe”.

Additionally, the request should not include the “hub.events” key-value pair and should add a “hub.channel.endpoint” key-value pair to its initial HTTP request body. The value for “hub.channel.endpoint” parameter should be the endpoint used for the subscription.

The rest of the handshake is the same, with the WSS result being sent through WebSocket as soon as the result can be determined, since the application has already connected in a previous subscription.

If the endpoint passed with the HTTP request is not an endpoint associated with an existing, unexpired subscription, the un-subscription will be denied.

Sending a valid un-subscription **removes** any original subscription associated with the access token and endpoint pairing used. It also closes the WebSocket connection, meaning the endpoint cannot be used for re-subscription in the future. If un-subscription is denied for any reason, the original subscription and connection will remain unaffected.

Example un-subscription request:

```
    POST https://subspace.example.com
    Authorization: Bearer i8hweunweunweofiwweoijewiwe
    Content-Type: application/x-www-form-urlencoded

    hub.topic=fdb2f928-5546-4f52-87a0-0648e9ded065&hub.mode=unsubscribe&hub.channel.typ
    e=webhook&hub.channel.endpoint=wss%3A%2F%2Fsubspace.example.com%2Fsession%2Fws%2Fv7
    tfwuk17a
```

Subscription/Token Expiration & Revocation

When a subscription and/or access token associated with a subscription expires, the Subspace server will send a subscription denial message via the open WSS connection. The denial message will be sent when Hyperdrive recognizes that a subscription/token is expired, which may not be immediately after the expiration happens. This may also happen if the access token used to subscribe gets revoked for any reason.

Once the denial message is sent to the application, the subscription is considered canceled and a new Subscription Handshake must be completed to restart the integration. The endpoint from the original subscription may not be reused.

Example expiration message:

```
{
    "hub.mode": "denied",
    "hub.topic": "fba7b1e2-53e9-40aa-883a-2af57ab4e2c",
    "hub.events": "patient-open,patient-close,com.epic.shutdown",
    "hub.reason": "Access token has expired."
}
```

Message Passing

Once a Subspace integration has been initialized, the integration is mediated through messages passed over HTTPS and WSS. Each message is associated with a particular event (e.g. closing a patient's chart) that is identified by an event name (e.g. "patient-close"). Messages sent to Hyperdrive are generally requests for an event to occur. Messages from Hyperdrive are generally notifications that an event has occurred.

These semantics vary event-by-event and integration-by-integration; refer to your integration's spec document for details.

To Hyperdrive

A message sent to Hyperdrive via Subspace is called a Request for Context Change (RCC) per the FHIRcast pattern.

RCCs are sent over HTTPS to Hyperdrive's Subspace server.

Sending HTTP Request

To send an RCC to Hyperdrive, an application sends an HTTP POST request to Hyperdrive's Hub URL with the following headers:

Header Name	Value
Content-Type	"application/json"
Authorization	"Bearer <ACCESS-TOKEN>" where <ACCESS-TOKEN> is the OAuth 2.0 access token mentioned above.

The body of the request is a JSON object with the following properties:

Property Name	Value
id	a unique identifier for this event notification
timestamp	a UTC timestamp for when the event notification was sent out
event	a JSON object describing the event. See the "event" property description below

The "event" property has the following properties:

Property Name	Value
hub.topic	Hyperdrive's Hub Topic
hub.event	the event name for the event which for which the request was sent
context	varies event-by-event; refer to your integration's spec document for details

Example Request:

```
POST https://subspace.example.com/7jaa86kgdudewiaq0wtu
Authorization: Bearer i8hweunweunweofiwweoijewiwe
Content-Type: application/json

{
      "timestamp": "2018-01-08T01:37:05.14",
      "id": "q9v3jubddqt63n1",
      "event": {
            "hub.topic": "fdb2f928-5546-4f52-87a0-0648e9ded065",
            "hub.event": "patient-close",
            "context": [
                  {
                        "key": "patient",
                        "resource": {
                        "resourceType": "patient-close",
                        "id": "ewUbXT9RWEbSj5wPEdgRaBw3",
                        //...other properties...
```

```
                    }
                ]
            }
        }
```

Receiving HTTP Response

If the access token sent with the request is not associated with an unexpired subscription to the relevant event OR the subscription does not have "write" permission for the event, the Subspace server responds with 401 Unauthorized.

If the HTTP request is malformed in any way, the Subspace Server responds with 400 Bad Request.

If there are other event-specific issues with the request, the Subspace Server responds with 400 Bad Request; refer to your integration's spec document for details.

Otherwise, if the request is accepted and will be processed, the Subspace server responds with 202 Accepted.

A 202 Accepted result only indicates that the RCC could be processed, not that any action will be performed in response to it. Refer to your integration's spec document for details on any further responses you can expect from Hyperdrive in response to an RCC for a specific event.

From Hyperdrive

A message received from Hyperdrive via Subspace is called an Event Notification.

Event Notifications are sent over WSS to an application subscribed to an event.

Receiving WSS Message

An event notification is sent via a WebSocket message to an application only if that application has an open connection associated with an unexpired subscription for the relevant event AND that subscription has "read" permission for the event.

The notification is sent as a JSON object with the following properties:

Property Name	Value
id	unique identifier for this event notification
timestamp	a UTC timestamp for when the event notification was sent out
event	a JSON object describing the event. See below.

The "event" property has the following properties:

Property Name	Value
hub.topic	Hyperdrive's Hub Topic
hub.event	the event name for the event which prompted this notification

context	varies event-by-event; refer to your integration's spec document for details.

Example Notification:

```
{
        "timestamp": "2018-01-08T01:37:05.14",
        "id": "q9v3jubddqt63n1",
        "event": {
                "hub.topic": "fdb2f928-5546-4f52-87a0-0648e9ded065",
                "hub.event": "patient-close",
                "context": [
                        {
                                "key": "patient",
                                "resource": {
                                "resourceType": "patient-close",
                                "id": "ewUbXT9RWEbSj5wPEdgRaBw3",
                                //...other properties...
                        }
                ]
        }
}
```

Sending WSS Response

When an application receives an Event Notification, the application sends a confirmation via a WSS message. The message contains a single JSON object with the following properties:

Property Name	Value
id	the identifier for the received event notification
status	a numeric HTTP status code indicating the success (2XX) or failure of the event notification (4XX-5XX)

Example:

```
{
      "id": "q9v3jubddqt63n1",
      "status": "200"
}
```