



# FHIR Support in InterSystems Products

Version 2020.3  
2021-02-04

*FHIR Support in InterSystems Products*  
InterSystems Version 2020.3 2021-02-04  
Copyright © 2021 InterSystems Corporation  
All rights reserved.

InterSystems, InterSystems IRIS, InterSystems Caché, InterSystems Ensemble, and InterSystems HealthShare are registered trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Response Center (WRC)**  
Tel: +1-617-621-0700  
Tel: +44 (0) 844 854 2917  
Email: [support@InterSystems.com](mailto:support@InterSystems.com)

# Table of Contents

<b>About This Book</b>	<b>1</b>
<b>1 Productions</b>	<b>3</b>
1.1 Accepting FHIR Requests	3
1.1.1 Creating the Interoperability Production	3
1.1.2 Configuring the Endpoint	4
1.2 Sending FHIR Requests	4
1.3 Interoperability FHIR Client	5
1.4 Messaging	6
1.5 Transformations	6
1.6 Use Cases	7
<b>2 FHIR Clients</b>	<b>9</b>
2.1 Interactions and Operations	10
2.1.1 Calling an Interaction Method	10
2.2 Customizing Requests and Responses	10
2.3 Requests without FHIR Client Class	11
2.3.1 Direct Calls to <code>DispatchRequest</code>	11
<b>3 FHIR Server: An Introduction</b>	<b>13</b>
3.1 Architecture	13
3.1.1 More About the Service	14
3.1.2 More About the <code>InteractionsStrategy</code>	15
3.1.3 More about the <code>Interactions</code> Class	15
3.1.4 Messaging	15
3.2 Writing a FHIR Server Application	15
3.2.1 Pre-Built Functionality	16
3.2.2 Developing Custom Functionality	16
3.3 Controlling Server Behavior	16
3.3.1 <code>ConfigData</code> Object	16
3.4 Making REST Calls	17
<b>4 Installing and Configuring a FHIR Server</b>	<b>19</b>
4.1 Configuring a FHIR Server	20
4.2 Installing and Configuring Programmatically	21
4.2.1 Configuring the FHIR Server Programmatically	23
<b>5 Resource Repository</b>	<b>25</b>
5.1 What is Supported?	25
5.1.1 Interactions	25
5.1.2 Operations	28
5.2 Migrating from Legacy Resource Repository	28
<b>6 Profiling FHIR</b>	<b>31</b>
6.1 Modifying the Capability Statement	31
6.1.1 Retrieving the Capability Statement	31
6.1.2 Updating the Capability Statement	32
6.2 Extensions	32
6.3 Custom Metadata Sets	32
6.3.1 Updating a Custom Metadata Set	33

6.4 Custom Search Parameters .....	33
<b>7 Operations .....</b>	<b>35</b>
7.1 Writing Methods for Custom Operations .....	35
7.2 Adding the Operation to Capability Statement .....	36
<b>8 Server Security .....</b>	<b>39</b>
8.1 Basic Authentication .....	39
8.1.1 Adding Authorization Requirements .....	39
8.2 OAuth 2.0 Authorization .....	40
8.3 No Authentication .....	41
<b>9 Server Debugging and Maintenance .....</b>	<b>43</b>
9.1 Debugging the FHIR Server .....	43
9.2 Logging .....	43
9.2.1 Internal FHIR Server Logging .....	44
9.2.2 HTTP Request Logging .....	44
9.2.3 FHIR Test Utility .....	45
9.3 Maintaining the FHIR Server .....	45
<b>10 FHIR Data .....</b>	<b>47</b>
10.1 Accessing FHIR Payloads .....	47
10.1.1 Implementations Without a Production .....	47
10.1.2 Production-Based Implementations .....	47
10.1.3 Direct Calls to Interactions Class .....	48
10.2 FHIR Data and Dynamic Objects .....	48
10.3 Data Load Utility .....	50
<b>11 SDA-FHIR Transformations .....</b>	<b>51</b>
11.1 Transformation Business Processes .....	51
11.1.1 SDA to FHIR Productions .....	51
11.1.2 FHIR to SDA Productions .....	53
11.2 Transformation APIs .....	54
11.2.1 SDA to FHIR APIs .....	54
11.2.2 FHIR to SDA APIs .....	56
11.3 Understanding SDA-FHIR Mappings .....	58
11.3.1 Accessing the FHIR Annotations Tool .....	58
11.3.2 Mappings Overview .....	59
11.3.3 Mapping Details .....	59
11.3.4 Lookup Table Mappings .....	60
11.3.5 Mapping Conventions .....	60
11.4 Customizing Transformations .....	64
11.4.1 Implementing Custom DTLs .....	64
11.4.2 Customizing Transformation API Classes .....	65
11.4.3 Customizing Lookup Tables .....	67
<b>12 ObjectScript Applications .....</b>	<b>71</b>
12.1 Bypassing the Service .....	71
12.2 Validating FHIR Resources .....	71
<b>13 Legacy FHIR Technology .....</b>	<b>73</b>
13.1 Upgrading Legacy Transformations .....	73
13.1.1 Upgrading Transformation Productions .....	74

# About This Book

This book explains how to use the FHIR<sup>®</sup> components of InterSystems products for FHIR STU3 and later.

Chapters include:

- [Productions](#)
- [FHIR Clients](#)
- [FHIR Server: An Introduction](#)
- [Installing and Configuring a FHIR Server](#)
- [Resource Repository](#)
- [Profiling FHIR](#)
- [Operations](#)
- [Server Security](#)
- [Server Debugging and Maintenance](#)
- [FHIR Data](#)
- [SDA-FHIR Transformations](#)
- [ObjectScript Applications](#)
- [Legacy FHIR Technology](#)

For more information about the FHIR standard, see <http://www.hl7.org/fhir/>; for FHIR license and legal terms see <http://www.hl7.org/fhir/license.html>.



# 1

## Productions

InterSystems healthcare products provide built-in business hosts that you can use to create an interoperability production that accepts and/or sends out FHIR requests. For example, there is a business service that takes in FHIR requests from the REST handler of an InterSystems FHIR endpoint and business operation that sends FHIR to an endpoint. If you are unfamiliar with interoperability productions, see [Introduction to Interoperability Productions](#).

To explore some of the FHIR implementations that are possible using an interoperability production, see [Use Cases](#).

**Note:** The InterSystems FHIR server does not require an interoperability production; by default, FHIR requests received by an endpoint's REST handler are sent directly to the FHIR server's [Service](#).

### 1.1 Accepting FHIR Requests

The built-in business service `HS.FHIRServer.Interop.Service` is designed to receive FHIR requests that have been sent to the endpoint that was created when you installed a FHIR server. Once configured, the endpoint's REST handler routes the request to `HS.FHIRServer.Interop.Service` rather than the FHIR server's `Service`. You need to install a FHIR server endpoint even if the FHIR request never reaches the InterSystems FHIR server's architecture, as is the case when the production's business operation forwards the request to an external system or the FHIR request is transformed into a different healthcare format.

Setting up an endpoint to route FHIR requests through an interoperability production is a two-step process:

- Create an interoperability production and add the `HS.FHIRServer.Interop.Service` business service.
- Configure an endpoint's **Service Config Name** field so it specifies the name of the business service that has been added to the interoperability production.

These steps can be taken in any order as long as, when the setup is complete, the name of the business service in the endpoint's configuration matches the name in the interoperability production.

#### 1.1.1 Creating the Interoperability Production

When the Foundation namespace for the FHIR server endpoint is created, the installation process also creates an interoperability production that should be used as the FHIR production. You need to modify the production to add the required business service that the endpoint uses to route requests through the production.

Interoperability productions that receive FHIR requests from the REST handler must include the `HS.FHIRServer.Interop.Service` business service. You can give the business service a custom name, but make sure that name matches the one specified for the endpoint's **Service Config Name** option.

## 1.1.2 Configuring the Endpoint

After [installing a FHIR server endpoint](#), the endpoint can be configured to use an interoperability production at anytime, including before the production has been created. Specifying the name of the business service while configuring the endpoint does not automatically create the business service in your production.

To configure an existing endpoint so FHIR requests are routed through a production:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Select the endpoint.
3. Select **Edit**.
4. In the **Service Config Name** field of the **Interoperability** section, specify the name of the business service of the production through which FHIR requests will be routed. For example, if the business service does not have a custom name, specify `HS.FHIRServer.Interop.Service`
5. Select **Update**.

## 1.2 Sending FHIR Requests

Within an interoperability production, business operations are responsible for making sure a FHIR request is sent to a FHIR endpoint. This request can originate from a variety of sources, for example, from an external FHIR client accessing an InterSystems endpoint or from a business process that transforms HL7 messages into FHIR requests. Regardless of its origin, there are two business operations available to send requests to a FHIR server:



Business Operation Class	Description
HS.FHIRServer.Interop.Operation	<p>Sends a FHIR request to the internal <a href="#">Service</a> of an InterSystems FHIR server in the local namespace. This business operation identifies the correct InterSystems FHIR server based on the URL of its endpoint, which is included in the <code>SessionApplication</code> property of the request message. If the message originated from a request sent to the FHIR server's endpoint through the REST Handler, the endpoint's URL is already part of the message. If the message was sent from the business process that transforms SDA to FHIR (<code>HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process</code>), the server is identified by the <b>FHIREndpoint</b> setting of the business process.</p>
HS.FHIRServer.Interop.HTTPOperation	<p>Sends a FHIR request to an internal or external FHIR endpoint over HTTP.</p> <p>If you are using a built-in business host to send the request to this business operation, use that business host's <b>TargetConfigName</b> setting.</p> <p>The default HTTP address of the FHIR endpoint is specified with the business operation's <b>ServiceName</b> setting, which refers to an entry in the <a href="#">Service Registry</a>. This default is overridden if a request includes an <code>AdditionalInfo</code> item named <code>ServiceName</code>, which specifies a Service Registry entry pointing to the alternate endpoint.</p>

If a *built-in* business host (such as `HS.FHIRServer.Interop.Service`) sends a request message (`HS.FHIRServer.Interop.Request`) to the `HS.FHIRServer.Interop.HTTPOperation` business operation, the request is sent over HTTP without custom code. However, if a FHIR payload is *formulated within a custom business host* that needs to put the payload into a FHIR request, you should instantiate an [interoperability FHIR client](#) to send the message. Similarly, if your custom business host needs to retrieve FHIR data from an endpoint, your production should use the FHIR client.

## 1.3 Interoperability FHIR Client

InterSystems technology provides a FHIR client object that simplifies the process of formulating a FHIR request from within a custom business host and sending it to a FHIR endpoint over HTTP. The business operation, `HS.FHIRServer.Interop.HTTPOperation`, that is used by the FHIR client to send the request over HTTP must be added to the interoperability production. Once the production is configured, your custom business host can use the FHIR client by instantiating `HS.FHIRServer.RestClient.Interop`, then calling the methods that correspond to FHIR interactions and operations.

Not all productions that send out FHIR requests over HTTP need to instantiate the interoperability FHIR client. For example, if SDA is being transformed into FHIR using `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process`, the FHIR forwarded from this business process to `HS.FHIRServer.Interop.HTTPOperation` is sent out via HTTP without

the FHIR client. However, when a FHIR payload is formulated by a custom business host within a production, the recommended method of sending it to a FHIR endpoint over HTTP is to instantiate the FHIR client.

When instantiating the FHIR client within the context of a custom business host, the call to the **CreateInstance()** method must contain the following parameters:

- **pServiceName** — Name of an entry in the [Service Registry](#) that points to a FHIR endpoint. This value overrides the **ServiceName** setting of the `HS.FHIRServer.Interop.HTTPOperation` business operation.
- **pTargetConfigName** — Name of the `HS.FHIRServer.Interop.HTTPOperation` business operation.
- **pHostObj** — Object instance of the business host that is instantiating the FHIR client. You can use `$this` to specify the current business host object that is instantiating the FHIR client.

For example, to instantiate a FHIR client within a business host with only the required arguments, enter:

```
Set fhirClient = ##class(HS.FHIRServer.RestClient.Interop).CreateInstance("MyFHIR.HTTP.Service", , ,  
, , , "HS.FHIRServer.Interop.HTTPOperation", $this)
```

The **CreateInstance()** method also accepts optional arguments that specify the value of the FHIR `prefer` header and send an OAuth token with the request.

Once the FHIR client has been instantiated, you can use it to send requests and perform operations. For details on using the FHIR client's methods to perform these actions, see [Interactions and Operations](#).

**Note:** The interoperability FHIR client class (`HS.FHIRServer.RestClient.Interop`) can also be used by a standalone ObjectScript application that needs to send a FHIR request through an interoperability production. In this case, the `HS.HC.Util.BusinessService` business service must be added to the production along with `HS.FHIRServer.Interop.HTTPOperation`. Instantiating the client is similar, but for standalone applications, the call to `CreateInstance` should not include an argument for the `pHostObj` parameter.

## 1.4 Messaging

The message class used to pass FHIR requests within the production is `HS.FHIRServer.Interop.Request`.

The message class used to pass a response from the FHIR server back through the production to the FHIR client is `HS.FHIRServer.Interop.Response`.

These classes include a property `QuickStreamId` that points to the FHIR payload. For information about working with a FHIR payload in JSON format, see [Accessing FHIR Payloads in Production-Based Implementations](#).

## 1.5 Transformations

You can add built-in business processes to your production to invoke SDA-FHIR transformations. For example, a production could consume HL7 messages, use a business process to convert the HL7 to SDA, and then use the built-in SDA-FHIR business process to convert the SDA to FHIR. To use the transformations, you must use the Installer Wizard to create a Foundation namespace, and then add the business process to the production that was created automatically when the namespace was created. No other production can be used. For more information about SDA-FHIR transformations using the built-in business processes, see [Transformation Business Processes](#).

## 1.6 Use Cases

The following use cases provide examples of how to use the built-in interoperability components to work with FHIR resources.

- [Proxy Server](#)
- [Transforming HL7 into FHIR](#)
- [Production-Based InterSystems FHIR Server](#)

### Proxy Server

InterSystems healthcare products can be used as a proxy server that accepts FHIR requests from an external FHIR client and forwards them to an external FHIR endpoint, then routes responses from the FHIR endpoint back to the external client. In this scenario, the FHIR client might be unaware that the InterSystems product is not the server that is accepting and producing FHIR, and the request or response can be manipulated within the production as needed.

You could implement a simple proxy server by:

- [Installing an InterSystems FHIR server](#) to create an endpoint. Even if your proxy server forwards the request to an external FHIR endpoint without accessing an internal InterSystems FHIR server, you still need to install the FHIR server to create the endpoint.
- Setting up the InterSystems endpoint to forward requests to `HS.FHIRServer.Interop.Service`. For more information, see [Accepting FHIR Requests](#).
- Adding `HS.FHIRServer.Interop.HTTPOperation` to the production and editing the **ServiceName** setting to specify the external FHIR endpoint.
- Editing the **TargetConfigName** of `HS.FHIRServer.Interop.Service` to point to `HS.FHIRServer.Interop.HTTPOperation`.

Of course, there are variations on the proxy server use case. For example, you could also add multiple `HS.FHIRServer.Interop.HTTPOperation` business operations and use a business process to determine which external FHIR endpoint should be the target of the proxy server. You could even add `HS.FHIRServer.Interop.Operation` to the production and have the proxy server store FHIR data in the internal InterSystems FHIR server along with sending it out to an external FHIR endpoint.

### Transforming HL7 into FHIR

InterSystems healthcare products simplify the process of extracting clinical data from incoming HL7 messages and transforming that data into FHIR resources. Once transformed into FHIR, the clinical data can be forwarded to external FHIR endpoints or stored in an internal FHIR repository that can be queried by FHIR clients. A basic interoperability production that transforms HL7 messages into FHIR resources would include:

- Adding a built-in business service that accepts HL7 messages into the production, for example, `EnsLib.HL7.Service.HTTPService`.
- Using a business host to transform the HL7 into SDA (the InterSystems intermediary data format). The following code added to a business process is enough to transform the HL7 into SDA:

```
do ##class(HS.Gateway.HL7.HL7ToSDA3).GetSDA(request,.con)
```

For more information about this transformation method, see [Data Transformations in InterSystems Healthcare Products](#).

- Adding the `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` business process to the production; this business process transforms SDA into FHIR.
- Modifying the **TargetConfigName** setting of the business host that contains the HL7-to-SDA transformation method to specify the name of `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process`.

Once the HL7 data has been transformed into FHIR, it can be sent to an external FHIR endpoint or, in the case of InterSystems IRIS for Health, stored in an internal Resource Repository of the FHIR server. You control where the FHIR data is forwarded by adding a business operation that performs a specific function. For details about these business operations, see [Sending FHIR Requests](#). If you are using the business operation that forwards requests to the internal storage of the FHIR server, use the **FHIREndpoint** setting of `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` to specify the InterSystems FHIR server's endpoint.

## Production-Based InterSystems Server

By default, requests to an InterSystems FHIR server do not go through an interoperability production, however you may want to use a production in some cases. For example, you may want to use a production during development to leverage message tracing and other advantages of productions, then make a small modification to send requests directly to the server's Service when it goes live. In an alternate use case, you might want to manipulate the FHIR requests using a business process before they reach the InterSystems FHIR server.

In its simplest form, a production-based FHIR server consists of configuring the production as described in [Accepting FHIR](#), then adding `HS.FHIRServer.Interop.Operation` as described in [Sending FHIR Requests](#). Once both business hosts are added to the production, modify the **TargetConfigName** setting of `HS.FHIRServer.Interop.Service` to specify the name of the `HS.FHIRServer.Interop.Operation` business operation.

If your aim is to use a production during development, then switch to a FHIR server that sends a request directly to the Service, simply [reconfigure the Server's endpoint](#) by removing the value in the **Service Config Name** field when the server goes live.

# 2

## FHIR Clients

InterSystems products come with standard FHIR client classes that your standalone ObjectScript application or interoperability production can use to send a FHIR request to a FHIR REST endpoint over HTTP or to a local InterSystems FHIR server. The methods that your application uses to make the requests are the same regardless of which FHIR client class your application is using. In each case, after instantiating the client class that corresponds to your use case, the application calls the method that corresponds to a [FHIR interaction](#) or [operation](#).

You have three client classes to choose from:

FHIR Client	Description
HS.FHIRServer.RestClient.HTTP	Sends a FHIR request over HTTP to a FHIR endpoint. When instantiating the class, the URL of the FHIR server's endpoint is identified by an entry in the <a href="#">Service Registry</a> .
HS.FHIRServer.RestClient.FHIRService	Sends a FHIR request to the Service of an InterSystems FHIR server in the same namespace. When instantiating the class, the InterSystems FHIR server is identified by the server's endpoint (for example, <code>/fhirapp/fhir/r4</code> )
HS.FHIRServer.RestClient.Interop	Uses an interoperability production to send a FHIR request over HTTP to a FHIR endpoint. It has two variations: <ul style="list-style-type: none"><li>• Send out a FHIR payload that has been formulated within a custom business host or retrieve FHIR data from within a business host.</li><li>• Route a FHIR request from a standalone ObjectScript application through an interoperability production before being sent over HTTP.</li></ul> For details about this interoperability FHIR client, see <a href="#">Interoperability FHIR Client</a> .

These classes all inherit from a single base class, `HS.FHIRServer.RestClient.Base`, that contains the logic for the methods that a FHIR client uses to perform a FHIR interaction or operation. Each type of FHIR client is instantiated with a `CreateInstance` method.

## 2.1 Interactions and Operations

Within the RESTful architecture of the FHIR specification, a FHIR client works with resources on the server through interactions. A FHIR client developed with InterSystems technology provides methods that correspond to these interactions, allowing your ObjectScript code to perform an interaction with a single method call.

While the FHIR client provides at least one method for every interaction, it provides a single method regardless of which operation you are performing on the FHIR server. For details on invoking this method to perform an operation, see the [Operation\(\) Class Reference](#).

### 2.1.1 Calling an Interaction Method

If your FHIR client is writing to the server with interactions like `update`, it must use the `SetRequestFormat` method to specify the format of the payload being written to the server. Possible formats are `JSON`, `XML`, `Form`, `XPatch`, and `JPatch`. Similarly, your FHIR client can specify the preferred format of the resources returned by the FHIR server using the `SetResponseFormat`. Possible formats are `JSON` and `XML`.

Unless the request and response formats change for individual interactions, your application can set them once and have them applied to all interaction methods. For example, a standalone FHIR client sending requests to a FHIR server over HTTP might set the request and response formats immediately after instantiating the client.

```
Set clientObj = ##class(HS.FHIRServer.RestClient.HTTP).CreateInstance("MyFHIR.HTTP.Service")
Do clientObj.SetRequestFormat("JSON")
Do clientObj.SetResponseFormat("JSON")
```

Once the FHIR client class has been instantiated and the request and response formats set, the application can call methods that correspond to the FHIR interactions they want to perform on the server. To explore the FHIR interaction methods, including signatures, that are available to a FHIR client, refer to the [HS.FHIRServer.RestClient.Base Class Reference](#). Note that FHIR interactions that allow conditional actions have two different methods. For example, your application can call `Update` or `ConditionalUpdate` depending on whether the `update` interaction is conditional.

The data type of the payload that is passed as an argument is determined by the type of FHIR client that has been instantiated.

- For clients accessing a FHIR server over HTTP, the payload argument can be a string or stream.
- For clients accessing an InterSystems FHIR server in the local namespace, the payload argument can be a string, stream, or dynamic object.

The following is an example of instantiating a FHIR client and performing a `read` interaction on the external FHIR server:

```
Set clientObj = ##class(HS.FHIRServer.RestClient.HTTP).CreateInstance("MyFHIR.HTTP.Service")
Do clientObj.SetResponseFormat("JSON")
Set clientResponseObj = clientObj.Read("GET", "Patient", "123")
```

## 2.2 Customizing Requests and Responses

Internally, each interaction method calls three overridable methods that can be customized to modify how a request is sent or to manipulate the response received by the request. These three methods, `MakeRequest`, `InvokeRequest`, and `MakeClientResponseFromResponse` are implemented by each type of FHIR client, not in the base class. Refer to the comments in the FHIR client class for more information (`HS.FHIRServer.RestClient.HTTP`, `HS.FHIRServer.RestClient.FHIRService`, or `HS.FHIRServer.RestClient.Interop`).

## 2.3 Requests without FHIR Client Class

Though using a FHIR client class is recommended when making requests to an internal FHIR server from an ObjectScript application, it is possible to write custom classes that perform CRUD operations on the server without these standard client methods. For example, you can write a custom class to interact with the FHIR server without going through the Service, thereby bypassing restrictions on the interactions that are allowed. For more information, see [Bypassing the Service](#). You can also make direct calls to the Service with the `DispatchRequest` method.

### 2.3.1 Direct Calls to `DispatchRequest`

An ObjectScript application can also act as a FHIR client by calling `DispatchRequest ( )` directly, which is the method used by the standard FHIR client and the internal FHIR server's REST handler.

#### 2.3.1.1 GET Resources

Your ObjectScript application can use the server's Service to retrieve resources. For example, assuming 178.16.235.12 is the IP address of InterSystems server and 52783 is the superserver port, a REST call might be:

```
GET http://178.16.235.12:52783/fhirapp/namespace/fhir/r4/patient/1
```

Using ObjectScript to access the same endpoint looks like:

```
set url = "/fhirapp/namespace/fhir/r4/"
set fhirService = ##class(HS.FHIRServer.Service).EnsureInstance(url)
set request = ##class(HS.FHIRServer.API.Data.Request).%New()
set request.RequestPath = "/Patient/1"
set request.RequestMethod = "GET"
do fhirService.DispatchRequest(request, .response)
```

In this example, the response is a data object (`HS.FHIRServer.API.Data.Response`) with the JSON response represented in a dynamic object.

**Note:** The first request to the server must instantiate the FHIR service by calling the `EnsureInstance` method. It does not cause problems to make this call before every request, but it takes a miniscule amount of time to check whether the service has been modified.

#### 2.3.1.2 POST Resources

You can also post data to the FHIR server programmatically. In the following example, suppose the application is creating a Patient resource that is described in a JSON object in the file `MyPatient.json`. The ObjectScript code might look like:

```
set url = "/csp/fhirapp/namespace/fhir/r4/"
set fhirService = ##class(HS.FHIRServer.Service).EnsureInstance(url)
set request = ##class(HS.FHIRServer.API.Data.Request).%New()
set request.RequestPath = "/Patient"
set request.RequestMethod = "POST"
set request.Json = {}.%FromJSON("c:\resources\MyPatient.json")
do fhirService.DispatchRequest(request, .response)
```

In this example, the source of the JSON stored in the request could have come from a dynamic object in your application rather than an external file.

#### 2.3.1.3 Handling FHIR Data as XML

When you use a REST client to perform CRUD operations on the FHIR server, the FHIR server automatically accepts or returns FHIR data as XML based on the incoming request. However, when you are performing CRUD operations programmatically from a custom class, all data going into the FHIR service must be in JSON format. Likewise, all data returned by the service is in JSON format. The FHIR server provides helper methods to convert XML to JSON and JSON to XML.

To send XML data into the FHIR service, put the XML into a stream object and send it to the

**HS.FHIRServer.Service.StreamToJSON()** method, specifying that the format is XML. For example, the following code turns the XML payload into a JSON request that can be passed to the FHIR service:

```
set url = "/csp/fhirapp/namespace/fhir/r4/"
set fhirService = ##class(HS.FHIRServer.Service).EnsureInstance(url)
set request = ##class(HS.FHIRServer.API.Data.Request).%New()
set request.Json= fhirService.StreamToJSON(MyStream,"XML")
```

To convert a JSON response from the FHIR service into XML, use the **HS.FHIRServer.Util.JSONToXML..JSONToXML()** method.

### 2.3.1.4 Handling FHIR Data as a Stream

The **HS.FHIRServer.Service.StreamToJSON()** method converts an XML or JSON stream into a JSON object so it can be passed to the FHIR service as part of a request. The FHIR service cannot handle a stream directly. The method accepts two arguments: the stream and the format of the data in the stream. For example, the following lines of code turn a JSON stream into a JSON object so it can be sent to the FHIR service:

```
set url = "/csp/fhirapp/namespace/fhir/r4/"
set fhirService = ##class(HS.FHIRServer.Service).EnsureInstance(url)
set request = ##class(HS.FHIRServer.API.Data.Request).%New()
set request.Json= fhirService.StreamToJSON(MyStream,"JSON")
```

For XML streams, simply pass XML as the second argument.



# 3

## FHIR Server: An Introduction

The following sections describe the basics of FHIR server technology in InterSystems products. For an overview of what a default FHIR server supports in InterSystems IRIS for Health, see [What is Supported?](#).

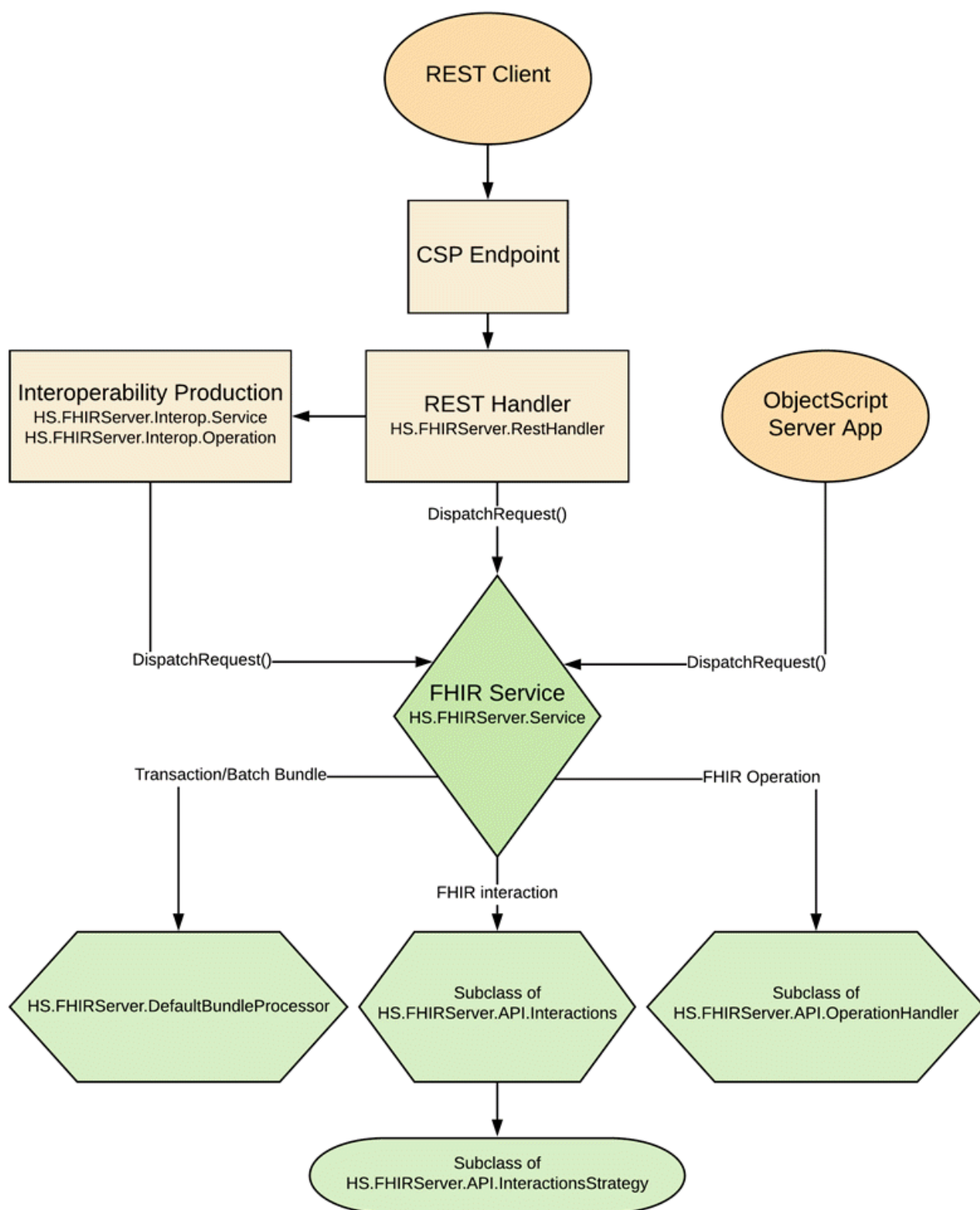
In most cases, implementing a FHIR server in Health Connect refers to the ability to accept requests from a FHIR client into an interoperability production. The default FHIR storage architecture provided with the FHIR server in InterSystems IRIS for Health is not available in Health Connect. Though it is possible to write a custom architecture for a FHIR server in Health Connect, most use cases do not include writing this custom code. For more information about using the endpoint of a FHIR server to receive FHIR requests in Health Connect, see [FHIR Productions](#).

### 3.1 Architecture

Tracing a FHIR request through the FHIR server provides a good overview of the major architectural features of the server. First, the FHIR request must reach the *Service*, which ensures that the request conforms to the server's FHIR metadata standards and then routes it to the appropriate component to handle the request. The FHIR request can reach this Service in three ways: from a REST handler, through an [interoperability production](#), or from a [server-side ObjectScript application](#). This Service is unrelated to a business service in an interoperability production.

What the Service does with the request depends on the type of request:

- If the request contains an HTTP method and endpoint that correspond to a [FHIR interaction](#), the Service forwards it to the method of the *Interactions* class that handles that type of FHIR interaction. For example, requests with a `read` interaction are sent to the `Read` method of the *Interactions* class. This *Interactions* class executes the FHIR interaction, using the *InteractionsStrategy* class to process the interaction according to the overall purpose of the FHIR server.
- For [FHIR operations](#), the Service forwards the request to a special class designed to perform operations. InterSystems IRIS for Health applications using the default Resource Repository offer out-of-the-box support for certain FHIR operations.
- If the request contains a [bundle](#) of type `transaction` or `batch`, the Service forwards the request to a special class that unpacks the bundle to perform the individual HTTP operations.



### 3.1.1 More About the Service

The Service is a singleton class that allows only one instance of itself to be instantiated for an endpoint. This instantiation occurs when the first FHIR request is sent to the Service by the REST Handler or Business Operation; once instantiated,

the Service exists until the process ends. For server applications making FHIR requests programmatically, the app must call `HS.FHIRServer.Service.EnsureInstance()` to retrieve the Service before sending the first request.

In most cases, the Service class (`HS.FHIRServer.Service`) is ready to uphold the endpoint's FHIR standard and route requests without being subclassed. Custom logic that determines how the FHIR server behaves is written into the `Interactions` and `InteractionsStrategy` subclasses, not the Service.

### 3.1.2 More About the InteractionsStrategy

The `InteractionsStrategy` class dictates the overall strategy for the FHIR server. It is the FHIR server application's backend, creating and implementing the environment in which the FHIR data is processed. When the installer creates a new endpoint, it calls the `Create` method of the `InteractionsStrategy` to set up the FHIR server for a specific purpose. The `InteractionsStrategy` superclass is `HS.FHIRServer.API.InteractionsStrategy`.

In many cases, the `InteractionsStrategy` is the "storage strategy" for how the FHIR server stores and retrieves FHIR resources. For example, the Resource Repository in InterSystems IRIS for Health is implemented by a subclass of `HS.FHIRServer.API.InteractionsStrategy` that creates the resource and index tables used to store and retrieve the FHIR data. In applications that are not storing FHIR data, the strategy might set up an environment that communicates with an external FHIR server or any other custom logic that works with the server's FHIR data.

The `InteractionsStrategy` is also responsible for managing the endpoints that have been registered in the namespace.

### 3.1.3 More about the Interactions Class

While the `InteractionsStrategy` class is the backend of the application, it uses the `Interactions` class to actually execute the FHIR interactions received by the Service. During this process, the `Interactions` class often calls methods in the `InteractionsStrategy` class, especially for structures and logic that are common to the entire FHIR server strategy. Because of their interdependent relationship, the `Interactions` class and `InteractionsStrategy` class are subclassed together in a unified approach. The `Interactions` superclass is `HS.FHIRServer.API.Interactions`.

The methods in the `Interactions` class that are called by the Service when processing a FHIR request can also be called directly from a server-side ObjectScript application. For example, a server-side application could call the `Add` method of the `Interactions` class rather than sending a POST request to the Service. In bypassing the Service, the server application can bypass any restrictions placed on the FHIR server by the Service's metadata. For example, the server application could populate the FHIR server's storage even though the endpoint is read-only for requests going through the Service.

The `Interactions` class also keeps track of which specialized classes the Service should use to perform FHIR operations, process bundles, and validate FHIR data. The Service obtains the name of these classes from the `Interactions` object when it needs to take action.

### 3.1.4 Messaging

The message class that the server architecture uses to pass FHIR requests is `HS.FHIRServer.API.Data.Request`.

The message class that the server architecture uses to pass responses from the server to the FHIR client where the request originated is `HS.FHIRServer.API.Data.Response`.

For information about accessing the FHIR payload in a message, see [Accessing FHIR Payloads](#).

## 3.2 Writing a FHIR Server Application

When developing an application that leverages the FHIR server technology, it is helpful to differentiate between classes that are intended to be subclassed and those that do not need to be modified.

## 3.2.1 Pre-Built Functionality

For most applications, the default Service (`HS.FHIRServer.Service`) effectively enforces the FHIR standard and routes requests for execution, and does not need to be subclassed. Likewise, the classes that process bundles (`HS.FHIRServer.DefaultBundleProcessor`) and validate resources (`HS.FHIRServer.API.ResourceValidator`) are fully-functional and do not need to be subclassed, though they can be if custom processing is required. In addition, the `ConfigData` object (`HS.FHIRServer.API.ConfigData`) that the Service uses to configure server behavior should not be subclassed.

InterSystems IRIS for Health comes with pre-built Interactions and InteractionsStrategy classes that store and retrieve resources from SQL tables, allowing you to implement a fully-operational FHIR server with minimal custom coding. This storage strategy is known as the [Resource Repository](#).

## 3.2.2 Developing Custom Functionality

Implementing custom functionality begins with subclassing the Interactions and InteractionsStrategy classes. While its possible to write a completely custom backend for your FHIR server by directly subclassing `HS.FHIRServer.API.Interactions` and `HS.FHIRServer.API.InteractionsStrategy`, if your application needs to store and retrieve FHIR resources, it is probably faster and more reliable to subclass the Resource Repository's `HS.FHIRServer.Storage.Json.InteractionsStrategy` and `HS.FHIRServer.Storage.Json.Interactions` classes to implement the custom logic. These classes inherit from the superclasses in the `HS.FHIRServer.API` package.

After using an IDE to create your Interactions and InteractionsStrategy subclasses, you must modify the following parameters of the InteractionsStrategy subclass:

- Modify the `StrategyKey` parameter to specify a unique identifier.
- Modify the `InteractionsClass` parameter to specify the name of your Interactions subclass.

Once you have successfully subclassed the Interactions and InteractionsStrategy, you can customize other aspects of the server's functionality like operations, bundle processing, and validation by subclassing the appropriate class (`HS.FHIRServer.API.OperationHandler`, `HS.FHIRServer.DefaultBundleProcessor`, and `HS.FHIRServer.API.ResourceValidator`, respectively). For details about FHIR operations, including adding custom ones without overwriting default operations included with the Resource Repository, see [FHIR Operations](#).

# 3.3 Controlling Server Behavior

The FHIR server installer takes a metadata set and InteractionsStrategy when creating an endpoint. While the InteractionsStrategy provides the backend logic of the FHIR server, the FHIR metadata set controls the capabilities of the server, for example what resources are allowed, what interactions on those resources are allowed, and the valid search parameters. InterSystems provides metadata sets for the base FHIR specifications of the supported FHIR versions; these default metadata sets should NEVER be modified directly. Rather, these standards can be extended or restricted by [creating a custom metadata set](#) or, in the special case of the [CapabilityStatement](#), by passing in a modified CapabilityStatement to the `SetMetadata` method of the Interactions class.

## 3.3.1 ConfigData Object

When the installer creates an endpoint, it also creates a `ConfigData` object (`HS.FHIRServer.API.ConfigData`) that keeps track of the InteractionsStrategy and metadata of the endpoint and specifies server configuration settings that affect things like security, session timeout, and search results. These configuration settings can be [specified with the Management Portal](#) or by [programmatically setting properties](#) of the `ConfigData` object.

The ConfigData object is managed by the InteractionsStrategy and used by the Service to configure itself.

## 3.4 Making REST Calls

When using a REST client to access the InterSystems FHIR server, keep the following in mind:

- The base path of an endpoint is: *ServerIPAddress:SuperServerPort/baseURL*, where:
  - *ServerIPAddress* is the IP address of the InterSystems server where the FHIR server is installed.
  - *SuperServerPort* is the InterSystems server's superserver port. You can find this superserver port in the Management Portal by going to **System Administration > Configuration > System Configuration > Memory and Startup**.
  - *baseURL* is the endpoint created during installation. For example, */fhirapp/namespace/fhir/R4*.

For example, a REST call to post a resource might look like:

```
POST http://178.16.123.19:52783/fhirapp/namespace/fhir/r4/Patient
```



# 4

## Installing and Configuring a FHIR Server

The Management Portal provides a **Server Configuration** page that allows you to install a new FHIR server and then configure it. Alternatively, you can [install and configure a server programmatically](#).

The FHIR server must be installed in a Foundation namespace; multiple FHIR servers can be installed in the same Foundation namespace.

**Important:** Before installing a FHIR server, you must consider whether you want to customize it now or in the future. If you install a FHIR server without creating new Interactions and InteractionsStrategy subclasses, you will not be able to perform certain customizations such as adding FHIR operations or modifying how bundles are processed. For information about preparing for these customizations before installing the FHIR server, see [Developing Custom Functionality](#). To view the options that can be configured without creating Interactions and InteractionsStrategy subclasses, see [Configuring a FHIR Server](#).

To install a new FHIR server from the Management Portal:

1. Open the Management Portal and switch to the Foundation namespace where you want the FHIR server installed. If you do not have a Foundation namespace, go to **Health**, and select **Installer Wizard** from the top menu bar. The **Configure Foundation** button allows you to create a new Foundation namespace. Be sure to activate the namespace after creating it.
2. Navigate to **Health > FHIR Configuration > Server Configuration**. If you do not see the **FHIR Configuration** menu, make sure you are using a Foundation namespace.
3. Select the **+** button in the **Endpoints** section.
4. Select the FHIR server's metadata set, which is based on a FHIR version. For example, if the server is working with FHIR R4 data, select **HL7v4**.

**Important:** If you plan to add custom search parameters for your endpoint now or in the future, you must create a custom metadata set first, then select it as the server's metadata set. For more information, see [Custom Metadata Sets](#).

5. Select the [InteractionsStrategy](#) for the endpoint. For InterSystems IRIS for Health, the default interactions strategy is the [Resource Repository](#) (HS.FHIRServer.Storage.Json.InteractionsStrategy), which stores FHIR data as JSON in dynamic objects.
6. Enter the base URL of the FHIR server's endpoint. The name of the URL must begin with a slash (/).
7. Select **Finish**.

## 4.1 Configuring a FHIR Server

Once you have installed a FHIR Server, you can configure its settings using the **Server Configuration** page of the Management Portal. These configuration settings can also be [modified programmatically](#) by setting the properties of the server's ConfigData object.

To configure the FHIR server:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Choose the endpoint of the FHIR server that you are configuring.
3. When the page expands, scroll down and select the **Edit** button.
4. Configure the settings, using the following descriptions as a guide.

Setting	Description
<b>Enabled</b>	Specify whether the endpoint is enabled. A disabled endpoint rejects requests from FHIR clients.
<b>Default Search Page Size</b>	Search result page size to use when a search does not contain a <code>_count</code> parameter.
<b>Max Search Page Size</b>	Maximum search result page size to prevent an excessive user-specified page size.
<b>Max Search Results</b>	Maximum number of resources that can be selected by a search before the server responds to the query with an error. This number only includes resources selected by the actual search; it does not include resources included via an <code>_include</code> search parameter. This value does not affect the size of pages returned by a search. Overly broad searches that select large numbers of resources take a lot of system resources to fulfill, and are probably more broad than the client actually needs.
<b>Max Conditional Delete Results</b>	Maximum allowable number of resources to delete via conditional delete. If the conditional delete search finds more than this number of resources, then the conditional delete as a whole is rejected with an HTTP 412 Precondition Failed error.
<b>FHIR Session Timeout</b>	Maximum number of seconds between requests to the service before any session data is considered stale.



Setting	Description
<b>Default Prefer Handling</b>	Specifies what happens by default when a search request contains an unknown parameter. Specify <code>lenient</code> to ignore the unknown parameter and return a bundle in which the <code>OperationOutcome</code> resource identifies the issue. Specify <code>strict</code> to reject the search request and return an error. A FHIR search request that includes the <a href="#">prefer header</a> overrides this default.
<b>OAuth Client Name</b>	Specifies the application name that the FHIR server, as an OAuth resource server, uses to contact the OAuth 2.0 authorization server when needed. For more information about OAuth 2.0 support, see <a href="#">OAuth 2.0 Authorization</a> .
<b>Required Resource</b>	If you specify an InterSystems security resource, FHIR clients must have privileges to the resource to perform interactions on the server. For more information, see <a href="#">Adding Authorization Requirements</a> .
<b>Service Config Name</b>	To route FHIR requests through an interoperability production before reaching the FHIR server, enter the package and name of the business service that will receive the requests. Unless the business service has a custom name, this entry is <code>HS.FHIRServer.Interop.Service</code> . For more details, see <a href="#">FHIR Productions</a> .
<b>Allow Unauthenticated Access</b>	Allows all FHIR requests to reach the server, ignoring authentication and authorization strategies.
<b>New Service Instance</b>	Instantiates a new <code>Service</code> object for every FHIR request.
<b>Include Tracebacks</b>	The FHIR server responds to a FHIR request by sending a stack trace in an <code>OperationOutcome</code> resource.

## 4.2 Installing and Configuring Programmatically

For applications that need to install a FHIR server programmatically rather than using the Management Portal, the server must be installed first, then configured.

The FHIR server must run in a foundation namespace, therefore creating a foundation namespace is a prerequisite to installing the FHIR server. Once you have a foundation namespace, the following methods of `HS.FHIRServer.Installer` must be called in order:

HS.FHIRServer.Installer method	Description
<b>InstallNamespace()</b>	Prepares an existing foundation namespace for the FHIR Server; it does not create a new foundation namespace. If called without an argument, the installer assumes the active namespace is a foundation namespace and prepares it for the FHIR server.
<b>InstallInstance()</b>	<p>Installs an instance of a FHIR Service into the current namespace. This method requires the following arguments:</p> <ul style="list-style-type: none"> <li>• Unique URL of the FHIR endpoint. This is passed as the first argument. Be sure the URL begins with a slash (/).</li> <li>• Classname of the FHIR Server's <a href="#">InteractionsStrategy</a>. This is passed as the second argument.</li> <li>• The identifier of the FHIR metadata set used for the FHIR server. This is the third argument. To use base FHIR metadata, pass HL7v30 for STU3 or HL7v40 for R4. If you want the FHIR server to use a custom metadata set, call <b>InstallMetadataSet()</b> first, then pass its key into <code>InstallInstance()</code>. For more information about creating a custom metadata set, see <a href="#">Custom Metadata Sets</a>.</li> </ul> <p>Two optional arguments can also be passed to <code>InstallInstance()</code>:</p> <ul style="list-style-type: none"> <li>• If the FHIR server is using OAuth to control authentication and authorization, enter the OAuth client name as the fourth argument. For more information about using OAuth security, see <a href="#">OAuth 2.0 Authorization</a>.</li> <li>• To make the endpoint read-only, pass 1 as the fifth argument. Default is read/write (0).</li> </ul>

For example, the following ObjectScript code installs a FHIR server using the default R4 metadata set and the default JSON storage strategy for InterSystems IRIS for Health (Resource Repository).

```

Set appKey = "/MyFHIRApp/fhir/r4"
Set strategyClass = "HS.FHIRServer.Storage.Json.InteractionsStrategy"
Set metadataConfigKey = "HL7v40"

//Install a Foundation namespace and change to it
Do ##class(HS.HC.Util.Installer).InstallFoundation("FHIRNamespace")
Set $namespace = "FHIRNamespace"

// Install elements that are required for a FHIR-enabled namespace
Do ##class(HS.FHIRServer.Installer).InstallNamespace()

// Install an instance of a FHIR Service into the current namespace
Do ##class(HS.FHIRServer.Installer).InstallInstance(appKey, strategyClass, metadataConfigKey)

```

## 4.2.1 Configuring the FHIR Server Programmatically

The `InstallInstance()` method creates a `HS.FHIRServer.API.ConfigData` object when it creates the FHIR server endpoint. You can control the behavior of the FHIR server by modifying the properties of this object. Refer to the class reference or the [configuration utility's settings](#) for a description of these properties.

For example, the code to change the debug mode of the FHIR server would retrieve the FHIR server's `configData` object, modify the `DebugMode` property, and then save the `ConfigData` object. The ObjectScript code might look like:

```
Set appKey = "/MyFHIRApp/fhir/r4"
Set strategy = ##class(HS.FHIRServer.API.InteractionsStrategy).GetStrategyForEndpoint(appKey)
Set configData = strategy.GetServiceConfigData()
Set configData.DebugMode = 4
Do strategy.SaveServiceConfigData(configData)
```



# 5

## Resource Repository

The Resource Repository is the default InteractionsStrategy for InterSystems IRIS for Health, allowing you to install a fully functioning FHIR® server without further development tasks. It automatically stores FHIR data received by the server as dynamic objects that encapsulate the JSON data structures of the FHIR data. Of course, you can extend the Resource Repository's classes, `HS.FHIRServer.Storage.Json.Interactions` and `HS.FHIRServer.Storage.Json.InteractionsStrategy`, to refine how the FHIR server handles the FHIR data. The Resource Repository also comes with default FHIR operations in the `HS.FHIRServer.Storage` package.

**Important:** The Resource Repository is not supported in Health Connect. Though you can create custom Interactions and InteractionsStrategy classes for a FHIR server in Health Connect, in most cases you are accepting FHIR into an interoperability production for other purposes. For more information, see [FHIR Productions](#).

For more information about programmatically retrieving data from or storing data in the Resource Repository, see [ObjectScript Applications](#). For more information about working with FHIR data once it is retrieved from the Resource Repository, see [Working with FHIR Data](#).

### 5.1 What is Supported?

When using the Resource Repository strategy provided with InterSystems IRIS for Health, the FHIR server supports the following interactions and operations. If your custom FHIR server extends the Resource Repository, it also supports these interactions and operations.

#### 5.1.1 Interactions

[FHIR interactions](#) are the set of actions that a FHIR client can take on resources. These interactions can be grouped according to whether they act upon an instance, a type, or the whole system. An instance is a specific instance of a resource, for example, `Patient/1` refers to an instance of a Patient resource with an `id` of 1. A type refers to a particular FHIR resource, for example, a Patient or Observation.

The following table summarizes the support for FHIR interactions in the Resource Repository, or a custom FHIR server that has extended the Resource Repository. If an interaction is not listed, it is not supported.

Interaction	Limitations/Notes
<a href="#">create</a>	Fully supported, including conditional create.
<a href="#">read</a>	The <code>_elements</code> parameter is not supported. Conditional read is not supported.
<a href="#">vread</a>	The <code>_elements</code> parameter is not supported. Conditional read is not supported.
<a href="#">update</a>	Fully supported, including conditional update.
<a href="#">delete</a>	Supported, but conditional delete is not supported.
<a href="#">history</a>	Supported for instance interactions only, not type or system. For example, <code>GET [baseURL]/Patient/1/_history</code> is supported, but not <code>GET [baseURL]/Patient/_history</code> or <code>GET [baseURL]/_history</code> . The <code>_count</code> and <code>_at</code> parameters are not supported. Paging is not supported.
<a href="#">batch</a>	Fully supported
<a href="#">transaction</a>	Circular references within the bundle are not supported.
<a href="#">search</a>	Supported with some limitations. For details, see <a href="#">Search Interaction</a> .

### 5.1.1.1 Search Interaction

FHIR clients use the search interaction to retrieve resources from the Resource Repository. For full details about the search interaction, refer to [FHIR specification](#). This section summarizes the default support for the search interaction when the FHIR server is using or extending the Resource Repository.

#### General Limitations

Keep in mind that a FHIR server using or extending the Resource Repository has the following limitations:

- Does not support searching across multiple resource types. For example `GET [base]?_id=1` is not supported.
- You cannot perform a search on all resource types within a compartment. For example, you cannot search for `[base]/Patient/10000001/?_id=008`. Therefore, searches within the context of a compartment must specify a resource type in that compartment. For example, you can use `[base]/Patient/10000001/Observation` to return all Observations in the specific patient's compartment or `[base]/Patient/10000001/Observation?status=final` to search for a subset of Observations within the compartment. If you want to retrieve a Patient's entire compartment, use the `$everything` operation (for example, `[base]/Patient/10000001/$everything`).

#### Search Parameter Types

Each search parameter has a [search parameter type](#) that determines how the parameter behaves. The following search parameter types are supported. If a search parameter type is not listed, it is not supported.

Parameter Type	Limitations/Notes
<a href="#">number</a>	Fully supported
<a href="#">date</a>	Fully supported
<a href="#">string</a>	Fully supported
<a href="#">token</a>	Does not support a token parameter with system value only ([parameter]=[system]   )
<a href="#">reference</a>	Fully supported
<a href="#">quantity</a>	Fully supported
<a href="#">uri</a>	Fully supported

## Parameters

The following [standard search parameters](#) are supported by the FHIR server when retrieving resource from the Resource Repository. If a parameter is not listed, it is not supported.

Parameter	Limitations/Notes
<code>_id</code>	Fully supported as described in the <a href="#">FHIR specification</a>
<code>_lastUpdated</code>	Fully supported as described in the <a href="#">FHIR specification</a>
<code>_tag</code>	Fully supported as described in the <a href="#">FHIR specification</a>
<code>_profile</code>	Fully supported as described in the <a href="#">FHIR specification</a>
<code>_security</code>	Fully supported as described in the <a href="#">FHIR specification</a>
<code>_source</code>	Fully supported.
<code>_has</code>	Fully supported as described in the <a href="#">FHIR specification</a>

## Modifiers

[Modifiers](#) can be added to the end of a parameter to affect the results of the search. The following modifiers are supported.

Modifier	Limitations/Notes
<code>:exact</code>	Supported for strings
<code>:contains</code>	Supported for strings
<code>:above</code>	Supported for uri
<code>:below</code>	Supported for uri
<code>:type</code>	Supported for references

## Prefixes

When using search parameters of type number, date, and quantity, you can add a [prefix](#) to the parameter's value to affect what resources match the search. For example, [parameter]=1e100 returns values that are less than exactly 100. The following prefixes are supported.

Prefix	Limitations/Notes
eq	Fully supported
ne	Fully supported
gt	Fully supported
lt	Fully supported
ge	Fully supported
le	Fully supported

### Search Result Parameters

[Search result parameters](#) help manage the resources returned by a search. The following search result parameters are supported. If a parameter is not listed, it is not supported.

Search result parameter	Limitations/Notes
_sort	Fully supported as described in the <a href="#">FHIR specification</a>
_count	Fully supported as described in the <a href="#">FHIR specification</a>
_summary	Supports <code>_summary=count</code> only. For details, see the <a href="#">FHIR specification</a> .
_include	Fully supported as described in the <a href="#">FHIR specification</a>
_revinclude	Fully supported as described in the <a href="#">FHIR specification</a>

## 5.1.2 Operations

For InterSystems IRIS for Health using or extending the default Resource Repository, the following operations are supported:

Operation	Limitations/Notes
\$everything	Fully supported
\$validate	<p>The validation modes (create, update, delete) are supported.</p> <p>Validation by profile is not supported.</p> <p>When a FHIR \$validate request includes a resource payload, the resource may be enclosed in a Parameters resource.</p>
\$lastn	Fully supported

## 5.2 Migrating from Legacy Resource Repository

For FHIR servers developed using InterSystems IRIS for Health 2019.4 or earlier, the data in the legacy Resource Repository must be migrated before using the new FHIR server architecture. To migrate your FHIR data:

1. Open the InterSystems Terminal and navigate to the namespace of your legacy FHIR server.



2. [Create a STU3 endpoint](#) that will work with the data in the existing Resource Repository.
3. Run the installation and configuration utility:  

```
do ##class(HS.FHIRServer.ConsoleSetup).Setup()
```
4. Choose option 6) Migrate Data from pre-2020.1.
5. Select the STU3 endpoint and confirm the migration.



# 6

## Profiling FHIR

According to the FHIR<sup>®</sup> specification, FHIR is a “platform specification” that requires modification to be suitable for a healthcare implementation and purpose. Adapting FHIR resources, frameworks, and API for a specific healthcare purpose is known as [Profiling](#). In this sense, FHIR profiles are just one part of profiling FHIR. The InterSystems FHIR server supports profiling FHIR in the following ways:

- [Capability Statement](#)
- [Extensions](#)
- [Search Parameters](#)

**Important:** When profiling, never modify the base metadata files installed with your InterSystems product; [create a custom metadata set](#) that extends this base metadata. As an exception, [modifying the capability statement](#) does not require creating a custom metadata set.

### 6.1 Modifying the Capability Statement

The FHIR server’s [capability statement](#) is client-facing metadata that documents how the server behaves; FHIR clients can retrieve the capability statement to determine what the server expects and how it will process FHIR requests. This capability statement can be edited to make small changes like changing the server’s name and description, or to describe functional changes like which resources the server accepts or what FHIR operations are available. In InterSystems products, updating the capability statement consists of retrieving it from the server, editing it, posting it back to the server.

**Note:** Do not update the capability statement when adding [custom search parameters](#). The capability statement is updated automatically when you update the custom metadata set that contains the new search parameters.

#### 6.1.1 Retrieving the Capability Statement

InterSystems strongly recommends retrieving the current capability statement from the server and modifying it rather than writing a new one. You can retrieve the capability statement resource with a REST client or programmatically. In the following examples, assume the IP address of the InterSystems server is 172.16.144.98, the superserver port is 52782, and the base url of the endpoint is /fhirapp/namespace/fhir/r4.

- To retrieve the capability statement with a REST client, send a GET request to `base-url/metadata`. For example:  
GET http://172.16.144.98:52782/fhirapp/namespace/fhir/r4/metadata

- To retrieve the capability statement programmatically, enter:

```
set strategy =  
##class(HS.FHIRServer.API.InteractionsStrategy).GetStrategyForEndpoint("/csp/fhirapp/namespace/fhir/r4")  
set interactions = strategy.NewInteractionsInstance()  
set capabilityStatement = interactions.LoadMetadata()
```

Once retrieved, the capability statement can be edited with an external editor or third-party tool. Capability statements retrieved programmatically must be converted from a dynamic object to a JSON file before modifying it.

## 6.1.2 Updating the Capability Statement

Once you have modified the capability statement, submit the revised version to the server programmatically from the InterSystems Terminal. In the following example, `/fhirapp/namespace/fhir/r4` is the endpoint's base url and `MyCapabilityStatement.json` is the revised version. The `{ }.%FromJson` method takes a JSON file and puts it into a dynamic object.

```
set strategy =  
##class(HS.FHIRServer.API.InteractionsStrategy).GetStrategyForEndpoint("/fhirapp/namespace/fhir/r4")  
set interactions = strategy.NewInteractionsInstance()  
set newCapabilityStatement = { }.%FromJson("c:\localdata\MyCapabilityStatement.json")  
do interactions.SetMetadata(newCapabilityStatement)
```

## 6.2 Extensions

The FHIR server accepts a resource with extensions as long as it is well-formed according to the FHIR syntax for extensions. For information about adding custom search parameters for an extension, see [Custom Search Parameters](#).

## 6.3 Custom Metadata Sets

By default, the installation process creates an endpoint based on the base metadata for a particular FHIR version. If you are planning to customize the FHIR server's metadata, for example by creating custom search parameters, create a custom metadata set before beginning the installation process, even if you plan to wait to implement the customizations.

**Note:** Though the server's capability statement is considered metadata, it is customized in a different manner. For more information, see [Modifying the Capability Statement](#).

To create a custom metadata set:

1. From the InterSystems Terminal, change to the FHIR server's namespace. For example:

```
set $namespace = "FHIRNamespace"
```

2. Run the installation and configuration utility:

```
do ##class(HS.FHIRServer.ConsoleSetup).Setup()
```

3. Choose option 8) Create a custom metadata set.
4. Choose the base FHIR metadata for the FHIR version of your endpoint. The custom metadata set extends this base metadata.
5. Enter a name of the metadata set. This name appears when you are installing a new endpoint.
6. Enter a description of the metadata set.

7. Enter the directory that contains or will contain the custom metadata. For example, this is the directory that contains or will contain JSON files with custom search parameters. If the directory you specified for the custom metadata is empty, you must update the metadata set once you add files to the directory.

Now that you have created a custom metadata set, it appears as an option when [installing an endpoint](#).

### 6.3.1 Updating a Custom Metadata Set

Whenever files are added to the custom metadata directory or the content of those files changes, you must update the metadata set so the server picks up the changes. The directory was specified when the custom metadata set was created. To update a custom metadata set:

1. From the InterSystems Terminal, change to the FHIR server's namespace. For example:

```
set $namespace = "FHIRNamespace"
```

2. Run the installation and configuration utility:

```
do ##class(HS.FHIRServer.ConsoleSetup).Setup()
```

3. Choose option 9) Update a custom metadata set.
4. Choose the custom metadata set you are updating.
5. If desired, enter a new description of the custom metadata set.
6. If desired, enter a new directory that contains the custom metadata.
7. When you see the prompt `Do you want to update the metadata cache?`, enter one of the following:
  - If you have added new metadata to the directory, choose `Y` to rebuild the search tables and add the metadata to the capability statement. Be aware that this can be a lengthy process.
  - If you only changed the description of the metadata, choose `N`.

## 6.4 Custom Search Parameters

The search parameters that a client can use to retrieve resources from the FHIR server are defined by [SearchParameter resources](#). In many cases, custom search parameters need to be added for custom extensions on resources.

To define custom search parameters, start by using an external editor or third-party tool to create JSON files that contain the new SearchParameter resources. Each JSON file must include a single bundle that contains one or more SearchParameter resources. You can define multiple files in which each bundle contains a single SearchParameter resource, or define a single file in which the bundle contains multiple SearchParameter resources.

The next step in the process depends on how the endpoint was installed:

- If the endpoint was installed using the base metadata rather than a custom metadata set, you will have to install a new endpoint. [Create a custom metadata set](#) that specifies the directory with the JSON files, and then install the new endpoint.
- If the endpoint was installed using a custom metadata set, place the JSON files in its custom metadata directory and [update the metadata set](#). You are given the option of specifying a new directory for the JSON files when you update the metadata set.
- If the endpoint has not been installed yet, [create a new metadata set](#) and specify the directory with the JSON files.

**Note:** You do not need to manually edit the server's capability statement when you add custom search parameters; the capability statement is automatically updated when you update the custom metadata set.

# 7

## Operations

The FHIR server supports FHIR [operations](#) that perform special functions based on requests from the FHIR client using an RPC-like approach rather than a RESTful one. These can be standard FHIR operations like `$everything` or custom ones. InterSystems IRIS for Health applications using the Resource Repository already support certain standard FHIR operations (see [What is Supported?](#) for a complete list). A FHIR server in Health Connect does not use the Resource Repository, so there are no default operations.

The following is an overview of the process of adding FHIR operations to your FHIR server.

1. Subclass the FHIR server's [Interactions](#) and [InteractionsStrategy](#) classes. If you are using the Resource Repository with InterSystems IRIS for Health, you want to subclass `HS.FHIRServer.Storage.JSON.Interactions` and `HS.FHIRServer.Storage.JSON.InteractionsStrategy`.
2. Create a subclass of `HS.FHIRServer.API.OperationHandler`. If you are using the [Resource Repository](#) that comes with InterSystems IRIS for Health, subclass `HSFHIRServer.Storage.BuiltInOperations` instead of `HS.FHIRServer.API.OperationHandler` so you do not lose the default operations like `$everything`. As a best practice, you might want to create a separate subclass for each operation, and then create a master class that inherits from all of them.
3. In your Interactions subclass, override the value of the `OperationHandlerClass` parameter to be the classname of the operation subclass that you just created.
4. [Write a method](#) for each operation in your operation handler subclass.
5. [Add the operations to the CapabilityStatement resource](#).

The following sections provide more details on the last two steps of the process.

## 7.1 Writing Methods for Custom Operations

Operations supported by the FHIR server correspond directly to methods in the operation handler subclass. The names of these methods must conform to the following syntax:

`FHIRScopeOpOperationName`

Within this syntax, the variables are:

- *Scope* identifies the type of endpoint to which the FHIR client is appending the operation. Possible values are:
  - *System* — Identifies operations that are appended to a “base” FHIR endpoint (for example, `http://fhirserver.org/fhir`). These operations apply to the entire server.

- **Type** — Identifies operations that are appended to a FHIR endpoint with a resource type (for example, `http://fhirserver.org/fhir/Patient`). These operations work with all instances of the specified resource type.
  - **Instance** — Identifies operations that are appended to a FHIR endpoint that points to a specific instance of a resource (for example, `http://fhirserver.org/fhir/Patient/1`). These operations work solely with a specific instance of a resource.
- *OperationName* is the \$ operation that the FHIR client appends to its call to the server.

The following table of examples shows the correlation between method names and the operations called by a FHIR client.

Method name	REST client call to the operation
FHIRSystemOpMyoperation	<code>http://fhirserver.org/fhir/\$myoperation</code>
FHIRTypeOpValidate	<code>http://fhirserver.org/fhir/Observation/\$validate</code>
FHIRInstanceOpEverything	<code>http://fhirserver.org/fhir/Patient/1/\$everything</code>

If your operation contains a hyphen (-), just remove the hyphen from the method name. For example, if the system-wide operation is \$my-operation, name the method `FHIRSystemOpMyoperation`.

The following is an example of the method signature for \$everything:

```
ClassMethod
FHIRInstanceOpEverything(pService
As
HS.FHIRServer.API.Service,
    pRequest
As
HS.FHIRServer.API.Data.Request,
    pResponse
As
HS.FHIRServer.API.Data.Response)
```

## 7.2 Adding the Operation to Capability Statement

The [capability statement](#) of the FHIR server must include all of the operations that the server supports. If the FHIR server is using a custom capability statement, [manually update the capability statement](#) to include the new operations.

Most FHIR server use a capability statement that has been customized in some way. However, if the FHIR server is using the default capability statement provided with your InterSystems product, you can use the following two-step procedure to automatically add a new operation to the capability statement.

1. Add the operation to the `AddSupportedOperations` method of the operation handler subclass. When the command-line utility generates the server's capability statement, it takes the supported operations from this method. As an example, the operation handling class for a server that supports the \$everything operations would include a method that looked like:

```
ClassMethod AddSupportedOperations(pMap As %DynamicObject)
{
    Do pMap.%Set("everything", "http://hl7.org/fhir/OperationDefinition/patient-everything")
}
```

If the superclass of your operation handling class already includes some operations, be sure to call the `AddSupportedOperations` method of that superclass within the `AddSupportedOperations` of the subclass. For example, the method of the operation handling subclass might look like:



```
ClassMethod AddSupportedOperations(pMap As %DynamicObject)
{
    Do ##class(HS.FHIRServer.MySuperclass.Validate).AddSupportedOperations(pMap)
    Do pMap.%Set("everything", "http://hl7.org/fhir/OperationDefinition/patient-everything")
}
```

If you created a subclass for each operation and a master class that inherits from all of them, make sure the master class calls the `AddSupportedOperations` method of each operation's subclass.

2. Use the command-line utility to re-generate the capability statement:

**Important:** Do not use the command-line utility to update the capability statement if the FHIR server is using a custom capability statement because the custom one will be overwritten. In these cases, [manually update the capability statement](#) with the new operations.

- a. From the InterSystems Terminal, change to the FHIR server's namespace. For example:

```
set $namespace = "FHIRNamespace"
```

- b. Run the installation and configuration utility:

```
do ##class(HS.FHIRServer.ConsoleSetup).Setup()
```

- c. Choose option 5) Update the CapabilityStatement Resource.
- d. Select the endpoint you are configuring.
- e. Specify whether you want the endpoint to be read-only.
- f. Confirm your selection.



# 8

## Server Security

You can control which clients can make requests to the FHIR<sup>®</sup> server and the interactions they can perform using [InterSystems security strategies](#) and [OAuth 2.0](#).

During development and debugging, you can temporarily [disable all security restrictions](#).

### 8.1 Basic Authentication

By default, the FHIR server enforces basic authentication in which any user with credentials to an InterSystems product can access the FHIR server by including those credentials in the header of the REST call. In this security strategy, the user's authorization within the InterSystems product is not a factor; any authenticated user can perform CRUD interactions on the FHIR server.

#### 8.1.1 Adding Authorization Requirements

By adding authorization requirements to basic authentication, you can restrict server access to InterSystems users who are authorized to work with a specific security resource (which is unrelated to a FHIR resource). In InterSystems security terms, only *users* who belong to *roles* that have *privileges* to the *resource* are authorized to perform interactions on the server. Users with a Write privilege to the required resource can perform create, delete, update, and conditional update interactions on the FHIR server. Users with a Read privilege to the resource can perform all interactions except the ones that require write access. Remember that FHIR transactions are recursive, so a user must hold Write privileges if the transaction request contains both read and write interactions.

The following is a basic overview of how to create a resource, assign privileges to the resource for a role, and assign users to the role. For a detailed description of InterSystems security, see the Security Administration Guide.

1. To create the resource that controls whether users are authorized to perform interactions on the server, open the Management Portal and navigate to **System Administration > Security > Resources**. Setting the **Public Permission** to Read allows all authenticated users to perform Read interactions on the server. For more information, see [Creating or Editing a Resource](#).
2. To create a role that will have privileges to the resource, navigate to **System Administration > Security > Roles**. Most commonly, there will be two roles, one for users who should have Read access and another for users who should have Write access. For more information, see [Creating Roles](#).
3. To grant privileges to a role:
  - a. Click **Add** in the **Privileges** section of the role's **General** tab.
  - b. Select the resource that will control server authorization, and click **OK**.

- c. Click **Edit** next to the new Privilege.
- d. Select the permissions you want the role to have for the resource.

For more information, see [Giving New Privileges to a Role](#).

4. Now that you have a role that has permissions to the security resource, select the **Members** tab and add the users that you want to have those permissions. For more information, see [Assigning Users or Roles to the Current Role](#).

### 8.1.1.1 Configuring the Server

Once you have created or chosen the security resource that will control a user's ability to perform FHIR interactions, use the following steps to configure the server to require this resource:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Select the endpoint of the FHIR server.
3. Select **Edit**.
4. In the **Required Resource** field, enter the name of the security resource that controls access to the FHIR server.
5. Select **Update**.

## 8.2 OAuth 2.0 Authorization

By setting up the FHIR server as an OAuth 2.0 resource server, you can reject a client's FHIR requests unless it has a valid access token that it obtained from the OAuth 2.0 authorization server. The first step in identifying the FHIR server as a resource server is to create a client configuration using **System Administration > Security > OAuth 2.0 > Client**. After creating a Server Description for the OAuth 2.0 authorization server, create a new client configuration for the FHIR server, specifying that it is of type Resource Server. For more information about setting up a resource server in InterSystems products, see [Using an InterSystems IRIS Web Application as an OAuth 2.0 Resource Server](#).

Once you have defined the client configuration for the FHIR server:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Select the endpoint of the FHIR server.
3. Select **Edit**.
4. In the **OAuth Client Name** field, enter the **Application Name** of the resource server as defined in the Management Portal.
5. Select **Update**.

**CAUTION:** The default FHIR server does not respect the scopes passed in an OAuth access token; by default, these scopes are ignored. InterSystems strongly recommends that you use InterSystems IRIS for Health to create an application or use an API management tool, such as InterSystems API Manager, that enforces scopes consistent with the FHIR standard to provide resource-based and identity-based access control. Failing to write custom code or implement another mechanism to handle scopes can result in privacy violations where FHIR clients can access or perform functions on data beyond or inconsistent with the scope that was specified in the access token.

## 8.3 No Authentication

While authentication is essential on a live FHIR server, being forced to provide credentials to the FHIR server during development and testing can be cumbersome. You can allow all FHIR requests to reach the server, temporarily ignoring authentication and authorization strategies. To allow unauthenticated access:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Select the endpoint of the FHIR server.
3. Select **Edit**.
4. Select the **Allow Unauthenticated Access** check box in the **Debugging** section.
5. Select **Update**.



# 9

## Server Debugging and Maintenance

### 9.1 Debugging the FHIR Server

Putting the FHIR server in debug mode helps solve problems during development and can temporarily eliminate the need to authenticate FHIR requests. To set debug options:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Select the endpoint of the FHIR server.
3. Select **Edit**.
4. In the **Debugging** section, select the check boxes of the debugging options you want to enable.
  - **Allow Unauthenticated Access** — Allows all FHIR requests to reach the server, ignoring authentication and authorization strategies.
  - **New Service Instance** — Instantiates a new Service object for every FHIR request.
  - **Include Tracebacks** — The FHIR server responds to a FHIR request by sending a stack trace in an `OperationOutcome` resource.
5. Select **Update**.

### 9.2 Logging

The FHIR server provides two types of logging:

- [Internal FHIR Server Logging](#) — Provides information about how the FHIR server architecture is processing FHIR requests, including which class methods are being called.
- [HTTP Request Logging](#) — Provides information about the HTTP requests coming from REST clients to the FHIR server.

## 9.2.1 Internal FHIR Server Logging

The FHIR server provides basic logging information about how the architecture is processing the FHIR requests being received by the server, including which class methods are being called, SQL-related messages, and how `_include` searches are being handled. To enable this type of logging:

1. Open the InterSystems Terminal.
2. Navigate to the FHIR server's namespace. For example, enter:

```
set $namespace = "FHIRNamespace"
```

3. Create a global, `^FSLogChannel`, that specifies what type of logging information should be stored. The syntax for creating the global is:

```
set ^FSLogChannel(channelType) = 1
```

Where *channelType* is one of the following:

- `Msg` — Logs status messages.
- `SQL` — Logs SQL-related information.
- `_include` — Logs information related to searches that use the `_include` and `_revinclude` parameters.
- `all` — Logs all three types of information.

For example, to enable logging for all types of information, enter:

```
set ^FSLogChannel("all") = 1.
```

**Note:** To switch to a new type of logging information (for example, from `Msg` to `SQL`), kill the existing `^FSLogChannel` global before setting it again with the new *channelType*.

### 9.2.1.1 Viewing the Log

Once logging for the FHIR server architecture is enabled, the log entries are stored in the `^FSLOG` global. To use the Management Portal to view the log, navigate to **System Explorer > Globals** and view the `FSLOG` global (not `FSLogChannel`). Make sure you are in the FHIR server's namespace.

Each node of the global is structured like:

```
CurrentMethod^CurrentClass|LogType|LogMessage
```

For example, a log entry in a node of the `^FSLOG` global might be:

```
"runQuery^HS.FHIRServer.Storage.Json.Interactions|SQL|Parameters: (2)"
```

### 9.2.1.2 Disabling Logging

To disable logging for the FHIR server architecture, simply kill the `^FSLogChannel` global or set it to 0. For example, you can enter the following in the Terminal:

```
kill ^FSLogChannel
```

## 9.2.2 HTTP Request Logging

When HTTP request logging is enabled, the REST handler that is receiving requests from FHIR clients writes information about each HTTP request to the `ISCLog` global. To enable this type of logging:



1. Open the InterSystems Terminal.
2. From any namespace, enter the following commands to configure the global ^%ISCLOG to start logging HTTP requests:

```
set ^%ISCLOG=5
set ^%ISCLOG("Category","HSFHIR")=5
set ^%ISCLOG("Category","HSFHIRServer")=5
```

### 9.2.2.1 Viewing the Log

Once logging for HTTP requests is enabled, the log entries are stored in the ^ISCLOG global, which is located in the %SYS namespace.

To use the Management Portal to view the log, navigate to **System Explorer > Globals** and view the ISCLOG global (not %ISCLOG). Make sure you are in the %SYS namespace.

### 9.2.2.2 Disabling Logging

To disable HTTP request logging, open the Terminal and enter the following command:

```
set ^%ISCLOG=1
```

## 9.2.3 FHIR Test Utility

The FHIR Test Utility that appears in the Management Portal (**Health > FHIR Test Utility**) does not work with the current FHIR architecture. It still works with the legacy FHIR technology.

## 9.3 Maintaining the FHIR Server

While maintaining a FHIR server that is in production, it might be necessary to stop processing FHIR requests to the endpoint, then re-enable the endpoint when the maintenance is complete.

To stop and re-start an endpoint:

1. In the Management Portal, navigate to **Health > FHIR Configuration > Server Configuration**. Make sure you are in the FHIR server's namespace.
2. Select the endpoint of the FHIR server.
3. Select **Edit**.
4. To make the FHIR server's endpoint available to requests, select the **Enabled** check box in the **Configuration** section. To stop an endpoint and reject requests, clear the check box.



# 10

## FHIR Data

Within the FHIR server architecture, FHIR data is represented in dynamic objects, so working with the data is a combination of knowing how to manipulate dynamic objects and how FHIR resources are represented in JSON. Consult the [FHIR specification](#) for details about JSON representations of FHIR resources. For details about manipulating FHIR data as dynamic objects, see [working with dynamic objects](#). When working with FHIR data in interoperability productions, the FHIR payload can be in formats other than JSON, so manipulating the data would not always involve dynamic objects.

### 10.1 Accessing FHIR Payloads

The process of accessing FHIR payloads varies depending on the message class carrying the payload. The messages classes for a FHIR production are different than the message classes of a default implementation that does not use a production, therefore accessing the FHIR payload of requests and responses varies depending on the implementation.

#### 10.1.1 Implementations Without a Production

By default, when a FHIR request is received by the REST handler, it stores the FHIR payload in the `Json` property of a Request object (`HS.FHIRServer.API.Data.Request`), which automatically puts the JSON structure into a dynamic object. FHIR requests that contain XML are converted to JSON before being represented as a dynamic object in the `Json` property. Responses from the FHIR server (`HS.FHIRServer.API.Data.Response`) also contain a `Json` property for FHIR data.

Working with FHIR data begins by getting access to the `Json` property of the request or response. For example, the following code demonstrates how an ObjectScript application can retrieve a Patient resource from the FHIR server and store it in a `patient` variable so it can be manipulated as a dynamic object.

```
set url = "/fhirapp/namespace/fhir/r4"
set fhirService = ##class(HS.FHIRServer.Service).EnsureInstance(url)
set request = ##class(HS.FHIRServer.API.Data.Request).%New()
set request.RequestPath = "/Patient/1"
set request.RequestMethod = "GET"
do fhirService.DispatchRequest(request, .response)
set myPatient = response.Json
```

For more information about storing requests and retrieving responses programmatically, see [ObjectScript Applications](#).

#### 10.1.2 Production-Based Implementations

When a FHIR implementation is using an interoperability production, you access the FHIR payload of the message object differently than implementations where a production is not used. In production-based implementations, the request and

response messages (HS.FHIRServer.Interop.Request and HS.FHIRServer.Interop.Response) contain a `QuickStreamId` that is used to access a `QuickStream` object containing the FHIR payload. Though an interoperability request message also contains a `Request` property of type `HS.FHIRServer.API.Data.Request`, this `Request` property cannot be used to access the FHIR payload because its `Json` property is transient (the same is true for interoperability responses). As a result, a business host in the production that needs to access the FHIR payload must use the `QuickStreamId` to obtain the payload.

If the payload is in JSON format, the business host can access the payload and convert it to a dynamic object in order to modify it. For example, a BPL business process could use the following code to access and modify the FHIR payload of a request message that is in JSON format:

```
//Identify payload as a Patient resource and convert to dynamic object
if ((request.Request.RequestMethod="POST") & (request.Request.RequestPath="Patient")){
    set stream = ##class(HS.SDA3.QuickStream).%OpenId(request.QuickStreamId)
    set myPatient = ##class(%DynamicObject).%FromJSON(stream)

    // Modify Patient resource
    do myPatient.%Set("active", 0, "boolean")

    //Update payload with modified Patient resource
    do
myPatient.%ToJSON(stream)
do stream.%Save()
}
```

### 10.1.3 Direct Calls to Interactions Class

FHIR data can be retrieved from the server's storage strategy programmatically by calling methods of the Interactions class (HS.FHIRServer.API.Interactions). This data is retrieved as a dynamic object. For more information about these method calls, see [Bypassing the Service](#).

## 10.2 FHIR Data and Dynamic Objects

Since FHIR data is represented as dynamic objects within InterSystems products, knowing how to work with dynamic objects is essential. The following code fragments provide an introduction to manipulating with dynamic objects that contain FHIR data. As you'll see, you need to be familiar enough with the [FHIR specification](#) to know the structure of fields in the JSON representation of a FHIR resource. For complete details on handling dynamic objects, see [Using JSON](#).

These code examples assume you have a variable `patient` that is a dynamic object containing a FHIR Patient resource.

#### Searching for a Value

The following code searches through identifiers of the Patient resource looking for a particular system using two different approaches. In order to write this code, you would need to be familiar enough with the FHIR specification to know that the JSON structure of a Patient resource contains an `identifier` that has a `system` name/value pair.

```
// Put JSON representation of Patient resource into a dynamic object
set patient = ##class(%DynamicObject).%FromJSON("c:\localdata\myPatient.json")

//Searching for a identifier with a specific system
set mySystem = "urn:oid:1.2.36.146.595.217.0.1"

//Approach 1: Use an Iterator
if $isobject(patient.identifier) {
    set identifierIterator = patient.identifier.%GetIterator()
    while identifierIterator.%GetNext(, .identifier) {
        if identifier.system = mySystem {
            write "Found identifier: " _ identifier.value,!
        }
    }
}
```

```
//Approach 2: Use a 'for' loop
if $isobject(patient.identifier) {
  for i=0:1:patient.identifier.%Size()-1 {
    set identifier = patient.identifier.%Get(i)
    if identifier.system = mySystem {
      write "Found identifier: " _ identifier.value,!
    }
  }
}
```

## Extracting a Value

The following code fragment extracts the family name from the Patient resource.

```
if $isobject(patient.name) && (patient.name.%Size() > 0) {
  set myFamilyname = patient.name.%Get(0).family
}
```

## Modifying a Value

The following code fragment sets the Patient resource's active field, which is a boolean, to 0.

```
do
  patient.%Set("active", 0, "boolean")
```

## Adding a New JSON Object

When you want to add a new JSON object to an existing dynamic object, you can choose whether to use an ObjectScript syntax or a JSON syntax. For example, the following code adds a new identifier to the patient, using two different approaches that have the same result.

```
set mySystem = "urn:oid:1.2.36.146.595.217.0.1"
set myValue = "ABCDE"

// Approach 1: Use JSON syntax
if '$isobject(patient.identifier)' {
  set patient.identifier = ##class(%DynamicArray).%New()
}

do patient.identifier.%Push({
  "type": {
    "coding": [
      {
        "system": "http://terminology.hl7.org/CodeSystem/v2-0203",
        "code": "MR"
      }
    ]
  },
  "system": (mySystem),
  "value": (myValue)
})

//Approach 2: Use ObjectScript syntax
set identifier = ##class(%DynamicObject).%New()

set typeCode = ##class(%DynamicObject).%New()
set typeCode.system = "http://terminology.hl7.org/CodeSystem/v2-0203"
set typeCode.code = "MR"

set identifier.type = ##class(%DynamicObject).%New()
set identifier.type.coding = ##class(%DynamicArray).%New()
do identifier.type.coding.%Push(typeCode)
set identifier.system = mySystem
set identifier.value = myValue

if '$isobject(patient.identifier)' {
  set patient.identifier = ##class(%DynamicArray).%New()
}
do patient.identifier.%Push(identifier)
```

## 10.3 Data Load Utility

The Data Load utility sends resources and bundles that are stored in a local system directory directly to the FHIR server with or without going over HTTP. The local FHIR data fed into the Data Load utility can be individual resources, bundles, or both, and can be expressed in JSON, XML, or both. A common use of this utility is feeding large amounts of synthetic data from open source patient generators into the FHIR server.

If getting data to the FHIR server as fast as possible is the objective, it is better to send it directly to the server without using HTTP. In this case, pass the `FHIRServer` argument to the Data Load utility along with the server's endpoint. For example, suppose the server's endpoint is `/fhirapp/fhir/r4` and the directory that contains FHIR bundles is `c:\localdata`. To run the Data Load utility, enter

```
Set status =  
##class(HS.FHIRServer.Tools.DataLoader).SubmitResourceFiles("c:\localdata","FHIRServer","/fhirapp/fhir/r4")
```

The utility should print `Completed Successfully` when it is done processing the files. If it does not, you can print any errors by entering `Do $SYSTEM.Status.DisplayError(status)`.

Alternatively, you can send all the bulk data over HTTP by passing `HTTP` along with the name of a Service Registry HTTP service. For more information about creating a HTTP service, see [Managing the Service Registry](#). For example, you could run:

```
Set status =  
##class(HS.FHIRServer.Tools.DataLoader).SubmitResourceFiles("c:\localdata","HTTP","MyUniqueServiceName")
```

The Data Load utility takes three optional arguments that control whether it displays progress, logs statistics, or limits the number of files in the directory that it will process. For details on these arguments, see

**`HS.FHIRServer.Tools.DataLoader.SubmitResourceFiles()`**

# 11

## SDA-FHIR Transformations

InterSystems provides transformations that convert SDA objects into FHIR resources (and vice-versa) using the data transformation language (DTL). SDA is an intermediary clinical format that makes it easier to go from one standard to another. For example, rather than transform HL7v2 to FHIR directly, a system can convert HL7v2 to SDA and then SDA to FHIR. For more information about SDA, see [SDA: InterSystems Clinical Data Format](#).

The bi-directional SDA-FHIR transformations can provide useful functionality in many different use cases, including:

- Taking content from an SDA-aware system and providing it to a FHIR system.
- Taking content from an SDA-aware system and storing it in a FHIR repository.
- Taking content from multiple SDA-aware systems and normalizing it for use or storage in a FHIR system.
- Taking content from a FHIR system and providing it to an SDA-aware system.

You have two options for invoking the DTL transformations that convert SDA objects into FHIR resources and vice-versa. You can invoke the DTL transformations by adding [abuilt-in business process](#) to an interoperability production, or you can call the [transformation APIs](#) directly, for example, from a custom business process.

### 11.1 Transformation Business Processes

You can use built-in business processes to invoke SDA-FHIR transformations in an [SDA to FHIR Production](#) or [FHIR to SDA Production](#). For example, a production could consume HL7 messages, use a business process to convert the HL7 to SDA, and then use the built-in SDA-FHIR business process to convert the SDA to FHIR.

For more information about the underlying transformation code used by the built-in business processes, see [Transformation APIs](#). These APIs can be called directly from a custom business process.

#### 11.1.1 SDA to FHIR Productions

A built-in business process, `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process`, can be added to a production to transform SDA objects and containers to FHIR bundles. To use the transformation, you must use the Installer Wizard to create a Foundation namespace, and then add the business process to the production that was created automatically when the namespace was created. No other production can be used.

Once added to the production, this business process:

- Accepts an SDA container as input and loops through each contained object.
- Converts the SDA container to FHIR content, in the form of a FHIR [Bundle](#) resource.

- Forwards the FHIR content to the business host specified by the `TargetConfigName` setting.
- Receives a response from the business host.
- Returns a response (based on what it received) to the business host that originally called it.

The business process in the SDA to FHIR production calls a method of the `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR` class to perform the transformation. For details about how this class handles the transformation, see [Transformation Details](#).

### 11.1.1.1 Adding the Business Process

To begin, open the Foundation production in the Production Configuration window of the Management Portal and add the `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` business process. Once added, you can modify the [business process settings](#) that impact the transformation. For an introduction to adding business processes to an interoperability production, see [First Look: Connecting Systems Using Interoperability Productions](#).

### 11.1.1.2 Business Process Settings

Settings of `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` that influence SDA to FHIR conversions include:

- `TargetConfigName` — Specifies the business host to which `HS.FHIR.FromSDA.DTL.Transaction.Process` sends its output. This setting is located in the **Basic Settings** section of the **Settings** tab in the Production Configuration window.
- `TransmissionMode` — Specifies how the business process transmits the FHIR bundle for further processing:
  - `transaction` — The business process sends the bundle of resources in a single interaction and the processing succeeds or fails for the whole of the bundle; if processing any single resource fails, processing for the other resources (and the entire bundle) stops. This is the default.
  - `individual` — The business process sends each resource from the bundle separately as its own interaction.

This setting is located in the **Additional Settings** section of the **Settings** tab in the Production Configuration window.

- `FullTransactionResponse` — If selected, the FHIR request message that this process sends is created with a "PREFER" header value set to "return=representation". Per the FHIR spec, this header indicates to a FHIR server that every created or updated resource should be returned in its entirety as it is saved (i.e., with any modifications applied by the server). Whether the server actually does this depends on the server. In general, this setting should be left unchecked except during debugging or if the FHIR client has a specific need to receive back the created/updated resources, as requesting this information is likely to increase response time from the FHIR server. This setting is located in the **Additional Settings** section of the **Settings** tab in the Production Configuration window.
- `FHIRFormat` — Specifies whether the content is in XML or JSON format. This setting is located in the **Additional Settings** section of the **Settings** tab in the Production Configuration window.
- `FormatFHIROutput` — Specifies whether or not content is formatted for readability. If selected, this setting has a performance impact, and as such should be enabled only during development and testing. This setting is located in the **Additional Settings** section of the **Settings** tab in the Production Configuration window.
- `CallbackClass` — Deprecated.
- `ValidResourceRequired` — Deprecated.
- `OutputToQuickStream` — If selected, the FHIR payload sent by this business process is placed in an `HS.SDA3.QuickStream` object, and the id of the `QuickStream` object is placed in the `QuickStreamId` property of the request message. If left unselected, the FHIR output from the transformation is placed in the `Payload` property of the request message. This setting is located in the **Additional Settings** section of the **Settings** tab in the Production Configuration window.
- `TransformClass` — Specifies name of the class that performs the transformation. If you subclass `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR` to customize the transformation behavior, you need to specify the name of that subclass.



- **FHIRMetadataSet** — Specifies the version of the outgoing FHIR based on a metadata set. All available metadata sets appear in the drop-down list.
- **FHIREndpoint** — Specifies the endpoint of a FHIR server. This setting is required if your business process is sending the outgoing FHIR to an `HS.FHIR.Server.Interop.Operation` business operation on its way to the FHIR server's Service.

### 11.1.1.3 Assigning a Patient ID

You can use the `AdditionalInfo` property of the SDA message sent to `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` to assign an ID to the Patient resource that is created by the SDA-FHIR transformation. When the SDA message contains an `AdditionalInfo` item named `PatientResourceId`, the transformation takes the value of `PatientResourceId` and assigns it to the `Id` field of the generated Patient resource.

**Note:** The underlying class, `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR`, used by the transformation business process contains a method that can be overridden to assign IDs to resources, including patient resources. For more information, see [Customizing Transformation API Classes](#).

### 11.1.1.4 Messages

The request message to `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` is either `Ens.Container` or `HS.Message.XMLMessage`.

There is no response message from `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process`. It returns a success or failure status instead.

## 11.1.2 FHIR to SDA Productions

A built-in business process, `HS.FHIR.DTL.Util.HC.FHIR.SDA3.Process`, can be added to a production to transform FHIR resources and bundles into SDA objects and containers. To use the transformations, you must use the Installer Wizard to create a Foundation namespace, and then add the business process to the production that was created automatically when the namespace was created. No other production can be used.

Once added to the production, this business process:

- Accepts a FHIR resource or bundle as input.
- Converts the FHIR content to an SDA container.
- Forwards the container to the business host specified by the `TargetConfigName` setting.
- Receives the response from the business host.
- Returns a FHIR response (based on what it received) to the business host that originally called it.

The business process in the SDA to FHIR production calls a method of the `HS.FHIR.DTL.Util.API.Transform.FHIRTosDA3` class to perform the transformation. For details about how this class handles the transformation, see [Transformation Details](#).

### 11.1.2.1 Adding the Business Process

To begin, open the Foundation production in the Production Configuration window of the Management Portal and add the `HS.FHIR.DTL.Util.HC.FHIR.SDA3.Process` business process. Once added, you can modify the [business process settings](#) that impact the transformation. For an introduction to adding business processes to an interoperability production, see [First Look: Connecting Systems Using Interoperability Productions](#).

### 11.1.2.2 Business Process Settings

Settings of `HS.FHIR.DTL.Util.HC.FHIR.SDA3.Process` that influence FHIR to SDA conversions include:

- **TargetConfigName** — Specifies the business host where the XMLMessage that includes the SDA3 stream is sent after it is transformed from FHIR by the DTL transformation. This setting is located in the **Basic Settings** section of the **Settings** tab in the Production Configuration window.
- **CallbackClass** — Deprecated.
- **OutputToQuickStream** — By default, the output of `HS.FHIR.DTL.Util.HC.FHIR.SDA3.Process` is an `HS.Message.XMLMessage` object that contains the SDA3 stream produced by the DTL transformation. If this setting is checked, the SDA3 stream is placed in a separate `HS.SDA3.QuickStream` object, and the `QuickStreamID` of the `QuickStream` object is placed in the `AdditionalInfoItem` property of the `XMLMessage`. If this setting is not selected, the SDA3 stream is placed in the `ContentStream` property of the `XMLMessage`. This setting is located in the **Additional Settings** section of the **Settings** tab in the Production Configuration window.
- **TransformClass** — Specifies name of the class that performs the transformation. If you subclass `HS.FHIR.DTL.Util.API.Transform.FHIRToSDA3` to customize the transformation behavior, you need to specify the name of that subclass.
- **FHIRMetadataSet** — Specifies the version of the incoming FHIR based on a metadata set. All available metadata sets appear in the drop-down list.

## 11.2 Transformation APIs

Your application has access to both [SDA to FHIR APIs](#) and [FHIR to SDA APIs](#).

### 11.2.1 SDA to FHIR APIs

The APIs that your code uses to transform SDA to FHIR are found in `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR`. Your application can call the [TransformStream](#) or [TransformObject](#) method, depending on whether the SDA is in a `%Stream.Object` or an SDA object.

Both of these methods return a transformation object (`HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR`) that contains the FHIR output in its `bundle` property, which is of type `%DynamicObject`. This `Bundle` contains all of the resources generated by the transformation with all references resolved.

Your code could serialize this `bundle` property from a dynamic object to JSON or XML with the following code. It assumes that `SDA3ToFHIRObject` is the transformation object returned by one of the transformation methods.

```
Set stream = ##class(%Stream.TmpCharacter).%New()
Set metadataSetKey = "R4"
If format="JSON" {
    Do SDA3ToFHIRObject.bundle.%ToJSON(stream)
} ElseIf format="XML" {
    Set schema = ##class(HS.FHIRServer.Schema).LoadSchema(metadataSetKey)
    Do ##class(HS.FHIRServer.Util.JSONToXML).JSONToXML(SDA3ToFHIRObject.bundle, stream, schema)
}
Do stream.Rewind()
```

#### 11.2.1.1 Using the [TransformStream](#) Method

The `TransformStream` method of `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR` takes in SDA as a `%Stream` and transforms it into a FHIR bundle. Its signature is:

```

TransformStream(stream
As
%Stream.Object, SDAClassname
As
%String,
    fhirVersion
As
%String, patientId
As
%String
= "",
    encounterId
As
%String
= "")

```

#### Parameters:

- `stream` — A `%Stream` representation of an SDA object or Container.
- `SDAClassname` — The classname for the object contained in the stream (for example, `HS.SDA3.Container`).
- `fhirVersion` — The version of FHIR produced by the transformation. For example, `STU3` or `R4`.
- `patientId` — If this optional parameter is specified, the `Id` field of the generated Patient resource will have the specified value.
- `encounterId` — If this optional parameter is specified, the `Id` field of the generated Encounter resource will have the specified value. This parameter is ignored if the `stream` parameter is a SDA Container because a Container can have multiple encounters, making it impossible to determine which FHIR Encounter should be given the specified resource id.

### 11.2.1.2 Using the `TransformObject` Method

The `TransformObject` method of `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR` takes in SDA as a container or object class and transforms it into a FHIR bundle. Its signature is:

```

TransformObject(source, fhirVersion
As
%String, patientId
As
%String
= "", encounterId
As
%String
= "")

```

#### Parameters:

- `source` — The SDA container or SDA object class that will be converted into FHIR.
- `fhirVersion` — The version of FHIR produced by the transformation. For example, `STU3` or `R4`.
- `patientId` — If this optional parameter is specified, the `Id` field of the generated Patient resource will have the specified value.
- `encounterId` — If this optional parameter is specified, the `Id` field of the generated Encounter resource will have the specified value. This parameter is ignored if the `stream` parameter is an SDA Container because a Container can have multiple encounters, making it impossible to determine which FHIR Encounter should be given the specified id.

### 11.2.1.3 Transformation Details

The following describes the default behavior of SDA to FHIR transformations. For an introduction to methods that can be overridden to customize transformation behavior, see [SDA to FHIR Overridable Methods](#).

- The incoming stream or object is broken down into individual streamlets, which are in turn transformed into `STU3` resources.

- By default, UUIDs are generated and assigned to the `fullUrl` field of the Bundle resource. In this case, the resource itself does not have an `Id`. If you would rather provide a resource id, override the `GetId` method. In this case, the value for `fullUrl` is `baseURL/resourceType/id` and the resource references are `resourceType/id`.
- The methods do not modify incoming URLs at all by default. This behavior can be overridden with the `GetBaseURL` method: for example, if you are posting to a specific repository, you can provide the URL prefix for the repository.
- Resources will contain references to other resources regardless of the mechanism used to assign IDs.
- Patient and Encounter references will be added to all available resources using the Patient and Encounter streamlets. Encounter references can be made successfully only if the `EncounterNumber` fields in the SDA streamlets are used. If they are empty, no references will be generated.
- In the case of shared resources such as Organization, Practitioner, or Medication, a hash of the first 32 kilobytes of each resource is added to a hash table. Each subsequent shared resource is checked for duplication by searching the hash table for a direct match. If a match is found, the resource will be marked as a duplicate. This behavior can be changed by overriding the `IsDuplicate` method.
- Each resource is validated before being added to the Bundle. If a resource fails validation, an error is thrown and processing stops, which means the Bundle is not returned. This default behavior can be changed by overriding the `HandleInvalidResource` method.
- When one or more SDA properties do not map to a FHIR resource field in the target schema, the transformation maps the SDA data to a FHIR extension. For more information, see [FHIR Extensions](#).
- For details about how a specific SDA object or property maps to the target FHIR resource or field, see [Understanding SDA-FHIR Mappings](#).

## 11.2.2 FHIR to SDA APIs

The APIs that your code uses to transform FHIR to SDA are found in `HS.FHIR.DTL.Util.API.Transform.FHIRTToSDA3`. This class contains multiple APIs that can be used to transform FHIR to SDA, depending on your use case.

In most cases, if your application needs to transform a single FHIR resource or bundle, it should call the class method `TransformStream` or `TransformObject`, depending on whether the FHIR is in a `%Stream.Object` or a dynamic object. However, in cases where you are transforming multiple FHIR bundles or resources in succession, it might be more efficient to instantiate the transformation class once and then call the `Transform` method multiple times.

All of these transformation methods return a transformation object (`HS.FHIR.DTL.Util.API.Transform.FHIRTToSDA3`) that contains the SDA output in its `container` property, which is of type `HS.SDA3.Container`. The transformation object's `object` property contains the last SDA container or object that was generated by the transformation. If the last input was a bundle, the `object` property is an SDA container; if the last input was an individual resource, `object` is an SDA object.

### 11.2.2.1 Using the `TransformStream` Method

The `TransformStream` method of `HS.FHIR.DTL.Util.API.Transform.FHIRTToSDA3` takes in a FHIR resource or bundle represented as a `%Stream` and transforms it into an SDA Container. Resource references are honored only if a FHIR bundle is passed to the method. Its signature is:

```
TransformStream(stream
As
%Stream.Object, fhirVersion
As
%String, fhirFormat
As
%String)
```

Parameters:

- `stream` — A %Stream representation of the FHIR resource or bundle.
- `fhirVersion` — The version of the FHIR resource or bundle being transformed. For example, “STU3” or “R4”.
- `fhirFormat` — Specifies the format of the FHIR resource or bundle. Acceptable values are “JSON” and “XML”.

### 11.2.2.2 Using the `TransformObject` Method

The `TransformObject` method of `HS.FHIR.DTL.Util.API.Transform.FHIRTosDA3` takes in a FHIR resource or bundle as a dynamic object and transforms it into an SDA Container. Resource references are honored only if a bundle is passed to the method. Its signature is:

```
TransformObject(source
As
%DynamicObject, fhirVersion
As
%String)
```

Parameters:

- `source` — The FHIR resource or bundle represented as a dynamic object.
- `fhirVersion` — The version of the FHIR resource or bundle being transformed. For example, “STU3” or “R4”.

### 11.2.2.3 Using the `Transform` Method

The `Transform` method of `HS.FHIR.DTL.Util.API.Transform.FHIRTosDA3` takes in a FHIR bundle as a dynamic object and transforms it into an SDA Container. Resource references are honored only if a bundle is passed to the method.

`Transform` is the method called by the classmethods that transform FHIR into SDA. You might want to call it directly if you are transforming multiple FHIR resources in succession so you do not need to instantiate a `HS.FHIR.DTL.Util.API.Transform.FHIRTosDA3` object everytime. For example, the following code would transform a Patient resource, Encounter resource, and Observation resource using the same transformation object:

```
set r4schema = ##class(HS.FHIRServer.Schema).LoadSchema("R4")
set transformer = ##class(HS.FHIR.DTL.Util.API.Transform.FHIRTosDA3).%New(r4Schema)
do transformer.Transform(patient)
do transformer.Transform(encounter)
do transformer.Transform(observation)
```

The signature of the `Transform` method is:

```
Transform(source
As
%DynamicObject)
```

Parameters:

- `source` — The FHIR resource or bundle represented as a dynamic object.

### 11.2.2.4 Transformation Details

The following is an overview of the default behavior of FHIR to SDA transformations. For an introduction to methods that can be overridden to customize transformation behavior, see [FHIR to SDA Overridable Methods](#).

- An incoming FHIR Bundle is broken down into individual resources, and those resources transformed into SDA3 streamlets.
- If a resource referenced by another resource within the incoming FHIR bundle is not present in the bundle, the transformation of the bundle continues. To change this behavior, override the [HandleMissingResource](#) method.
- When a transformation is attempting to convert a reference to an object, no object will be created in the SDA streamlet if:

- A subtransformation exists but the referenced resource has no values for any of the elements with mappings.
- There is no subtransformation from the referenced resource type to the datatype in the SDA3 object.
- The `EncounterNumber` field on an Encounter streamlet will be populated starting at 1 and incremented for each encounter that is processed. Any subsequent resources that reference that Encounter resource, when transformed to SDA3, will perform a lookup based on the resource ID and will find the encounter number it should use. The assignment of encounter numbers can be overridden with `GetIdentifier` method. It can be useful to access the contents of the resource being converted in order to determine what `EncounterNumber` should be returned. The instance property `%currentReference` contains a FHIR reference object that can be passed into the instance method `GetResourceFromReference` in order to obtain the resource as a dynamic object.
- Similar to encounter numbers, `ExternalID` values for `HealthConcern` and `Goal` resources are populated starting at 1 by default. This behavior can be overridden with the `GetIdentifier` method.
- The value of the `SDA Container:SendingFacility` property is set as follows: if the Patient's `managingOrganization` field contains a reference to an `Organization`, and that `Organization` is in the Bundle, it is used. Otherwise, the patient identifiers are searched for an MRN with an assigning authority, and that assigning authority is used. If neither of these items is found, the string `FHIR` is used. This behavior can be overridden in the `GetSendingFacility` method.
- SDA3 extensions are not used. If a field does not exist in SDA3, the content will be dropped.
- If a Bundle comes in without a Patient resource, an error will be thrown. Other than that, no validation will be performed on the container. It will simply be returned as is.
- To view information about containment relationships, refer to the FHIR Annotations (**Health > FHIR Annotations**) in the Management Portal for the Bundle resource.
- For details about how a specific FHIR resource or field maps to an SDA object or property, see [Understanding SDA-FHIR Mappings](#).

## 11.3 Understanding SDA-FHIR Mappings

Whether you use the transformation API or a built-in business process to perform an SDA-FHIR transformation, you can use the **FHIR Annotations** tool to understand exactly how the SDA or FHIR data was transformed into the target format. The tool gives you an overview of which SDA object was mapped to a particular FHIR resource (or vice-versa) while providing the ability to drill down into the mapping to understand exactly how the properties of the SDA object resulted into fields of a FHIR resource (or vice-versa). When using the **FHIR Annotations** tool, SDA properties are referred to as fields, for example, mappings are referenced as being field-to-field rather than property-to-field. You can also explore how lookup tables were used to map codes between SDA and FHIR, learn more about the data types involved in the transformation, and discover which ObjectScript methods were used in the transformation.

To understand the logic behind mappings, see [Mapping Conventions](#).

### 11.3.1 Accessing the FHIR Annotations Tool

To access the **FHIR Annotations** tool:

1. Log in to the Management Portal as a user with the `%Ens_EDISchemaAnnotations` role.
2. Navigate to **Health** —> *FHIRnamespace*.
3. Expand the **Schema Documentation** menu option and click **FHIR Annotations**.

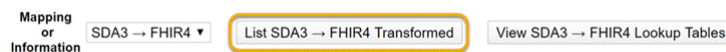
To begin exploring the mappings, use the **Mapping or Information** drop-down list to select the source and target of the transformations. For example, if you are interested in SDA3 to FHIR R4 mappings, select **SDA3 → FHIR4**.



While using the **FHIR Annotations** tool to explore the SDA-FHIR mappings, you can select the **Help** and **FAQ** buttons to obtain guidance on using and interpreting the user interface of the tool. In addition, hover text is available over many of the elements of the user interface.

## 11.3.2 Mappings Overview

Before drilling down into the details of a particular mapping (including field-to-field mappings), it can be useful to gain an overview of all the mappings between SDA objects and FHIR resources. To view a list of how objects and resources map to each other, select **List <transform> Transformed**.



## 11.3.3 Mapping Details

If you are interested in the details of how a specific SDA object or FHIR resource is mapped to the target format, you can select the object or resource from the drop-down lists. For example, to view the mapping of the Appointment resource to SDA3, select **Appointment** from the **FHIR4 by Name** drop-down list.










Each mapping is presented in a table that shows all of the SDA field-FHIR field mappings, cardinality of the source field, data type of the source field, and other useful information. To discover more details about the elements in the table, you can:

- Hover over each element in the table to obtain additional information.
- Click the links to open more details about that element, including the icons in the **Actions** column. For example, you can click a data type to explore how that data type is mapped.
- Click the Mapping Definition icon (🔍) to drill down into the technical details of the mapping. Once the Mapping Definition opens, you can click the FHIR data types to bring them up in the official FHIR specification. You can also view technical details like cardinality, default values, and the subtransformation or class method used by the mapping. In some cases, there are additional notes that help explain the mapping.

The following is a legend of the icons in the **Actions** column of the mapping table:



Icon	Meaning
	View the detailed Mapping Definition.
	Mapping uses a subtransformation or class method.
	Mapping uses a <a href="#">Condition to Set this Field</a> to control whether the mapping is used.
	Mapping uses a FHIR extension as its target.
	Mapping assigns a default value to the target when the source contains missing or invalid data.
	When the mapping is used, the transformation appends data to an existing target object, rather than creating a new target object.
	View mappings for the subfields of the FHIR resource field.

## 11.3.4 Lookup Table Mappings

The **FHIR Annotations** tool allows you to view the lookup tables that map codes between SDA and FHIR. For example, you can discover that the code A in the `Status` property of a `HS.SDA3.Alert` object maps to the `in-progress` code of the FHIR `event-status` value set. To explore the lookup tables used to map codes from the source to the target, select **View <transform> Lookup Tables**.



To customize a lookup table, see [Customizing Lookup Tables](#).

## 11.3.5 Mapping Conventions

This section explains the logic behind the SDA-FHIR mappings.

### 11.3.5.1 Field-to-Field Mappings

Most mappings are field-to-field: The mapping finds a data value in a source field and assign that value to a target field. For example, the value of a SDA property is assigned to a FHIR field.

### 11.3.5.2 Conditional Mappings

Some field-to-field mappings are conditional; the value is assigned to the target field only if certain conditions are met. The **FHIR Annotations** tool shows the label **Condition to Set this Field** when it presents this information. The DTL `<if>` element controls this in the code.

### 11.3.5.3 Literal Values

Among the defined mappings are mappings of literal values to target fields. One purpose of these mappings is to provide values for required target fields when the source object definition contains no fields that could provide the data required by the target.



Often, mappings of this type are defined conditionally, to be used only when needed.

### 11.3.5.4 Excluded Fields

SDA fields that contain metadata without clinical significance are not mapped to a FHIR field. For example, the `UpdatedOn` property is not transformed into FHIR.

In addition, SDA fields that are marked as Not Used in the class reference are not transformed into FHIR. For example, the `ExternalId` property of `LabResultItem` is not mapped to a field in the Observation resource.

### 11.3.5.5 Mapping Single to List

When the source field is single but the target field is a list, the transformation maps the source item to the first entry in a target list. After the transformation, the list contains only one entry. This feature is handled automatically during code generation for transformations. Single to List does not require special attention in the mapping definitions.

### 11.3.5.6 Mapping List to Single (Values)

When the source field is a list of values, and the target field is limited to a single value, the transformation concatenates the list of values into a single value, separating each value in the list with a semicolon and space.

### 11.3.5.7 Mapping List to Single (Objects)

For SDA to FHIR: when the incoming SDA is a list of objects, and the target FHIR has only one object, the mapping table contains two mapping entries for the source list field:

- One mapping maps the source list field to the target single field. The transformation generated from this mapping simply places the first list entry into the target field.
- The other mapping maps the source list field to the target FHIR extension that contains the full list of objects. The FHIR extension URL is the full source field name, including the resource name, but using all-lowercase text separated by hyphens.

For FHIR to SDA: when the incoming FHIR has a list of objects, and SDA has one object, the transformation uses the first object and drops all the others.

### 11.3.5.8 Mapping SDA `CodeTableDetail` to a FHIR Code

Transformations map an SDA `CodeTableDetail` (or one of its subclasses) to a FHIR coded object such as `Coding` or `CodeableConcept` as follows:

1. The Code value is mapped to the code field.
2. The Description is mapped to the display field.
3. If there is an `OriginalText` field, it is mapped to the text field.

### 11.3.5.9 Mapping Coded Values to FHIR using Lookup Tables

The mapping consults a lookup table to find the entry that maps code values from the source schema (SDA or FHIR) to code values in the target schema (FHIR or FHIR DSTU2) for this mapping.

If the mapping cannot find the lookup table, or cannot find a matching entry in the lookup table and it has a non-empty default value defined, it applies its default value to the code field. Otherwise, the target receives no value from this mapping.

If the mapping is SDA to FHIR, and the source field contains a non-empty value, then by convention there are two mapping entries for this source field. Both entries execute under the same Condition to Set this Field:

- One entry does the lookup to retrieve the value to assign to the target field.
- The other stores the original source field value in a string-valued FHIR extension.

In either case, if there is a Description or OriginalText along with the Code value, it is mapped to FHIR where applicable.

### 11.3.5.10 Mapping a FHIR Code to SDA CodeTableDetail

When a FHIR primitive code or coded object such as **Coding** or **CodeableConcept** does not use a lookup to transform the code value from FHIR to SDA, it is transformed to SDA **CodeTableDetail** (or one of its subclasses) as follows:

- **CodeableConcept.text** is transformed to **HS.SDA3.CodeTableTranslated.OriginalText**
- **CodeableConcept.coding.display** (or **Coding.display**) is transformed to **HS.SDA3.CodeTableDetail.Description**
- **CodeableConcept.coding.code** (or **Coding.code**, or simply **code**) is transformed to **HS.SDA3.CodeTableDetail.Code**
- **GetCodeforURI** of **CodeableConcept.coding.system** (or **Coding.system**) is transformed to **HS.SDA3.CodeTableDetail.SDACodingStandard**
- **CodeableConcept.coding.version** (or **Coding.version**) to **HS.SDA3.CodeTableDetail.CodeSystemVersionId**

### 11.3.5.11 Mapping FHIR Coded Values to SDA using Lookup Tables

If you want a mapping to use a code lookup table for FHIR to SDA, the mapping table contains two mapping entries for the source field:

- One of the two entries consults a lookup table to find the entry that maps a FHIR code value to an SDA Code.
- The other mapping entry in the pair takes over when the lookup table entry is unavailable or does not provide a match. It maps the source FHIR code value (unchanged) into an SDA **CodeTableDetail** object, as described above. That is, if the FHIR code was inside a **Coding** or **CodeableConcept** object, the FHIR code, display, system, version, and text values all are mapped appropriately into SDA **CodeTableDetail** fields.

### 11.3.5.12 Mapping SDACodingStandard

When the transformation encounters the **SDACodingStandard** property of an SDA object, it checks to see if the **SDACodingStandard** value is in the OID registry, and does one of the following:

- If the **SDACodingStandard** value is an entry in the OID registry that includes a URL, the transformation sets the **system** field of the FHIR Coding resource to the URL.
- If the **SDACodingStandard** value is an entry in the OID registry that does not define a URL, the transformation sets the **system** field of the FHIR Coding resource to the OID.
- If the **SDACodingStandard** value is not an entry in the OID registry, the transformation stores the value in a FHIR extension.

### 11.3.5.13 Mapping String Values to Numeric Values

When the target is FHIR, and a string value is mapped to a numeric value, the string may contain non-numeric text such as units of measurement or instructions. To handle this, there are two mapping entries for the source list field:

- One of the two entries always assigns the source string value to a FHIR extension that consists of one string-valued field.
- The other mapping entry tests the source string value to see if it is numeric. If so, it maps this numeric value to the target numeric field.

### 11.3.5.14 Multi-Part Literal Values for FHIR Code Objects

For some FHIR target fields that are **Coding** or **CodeableConcept** objects, a set of mappings from literal values forms a multi-part value that is assigned to the field when needed. The full set of fields that such an object can contain are: `code`, `system`, `display`, `text`, `version`, and `userSelected`.

Where this is the case, the DTL annotation element for the code field explains that this code resides within a **Coding** or **CodeableConcept** object that is receiving a multi-part literal value. The FHIR Annotations show that the set of literal value mappings relating to this code all have the same value in the Condition to Set this Field.

### 11.3.5.15 Mapping to FHIR Extensions

When the target of a transformation is FHIR, one or more SDA properties might not have a corresponding field in the target FHIR schema. In that case, transformations map the SDA data to a FHIR extension. The URL prefix for the extension is `http://intersystems.com/fhir/extn/sda3/lib`. The full URL is the full SDA property name, including the resource name, but using all-lowercase text separated by hyphens.

For example, the FHIR extension for the SDA property `HS.SDA3.Administration:AdministeredAmount` is:

- Extension name: `administration-administered-amount`
- Full URL for the FHIR extension:  
`http://www.intersystems.com/fhir/extn/sda3/lib/administration-administered-amount`

### 11.3.5.16 Mapping SDA CustomPairs

The transformations support the legacy `CustomPairs` property in SDA classes of type `HS.SDA3.SuperClass`.

`CustomPairs` is a collection of objects of type `HS.SDA3.NVPairs`, each of which has two properties, `Name` and `Value`. When the transformation code encounters this property in customer SDA data, and the target is FHIR, the collection is mapped to a FHIR extension that contains a `Parameters` resource. This `Parameters` resource is a collection of paired fields: `name` and `valueString`.

In the example below, the customized SDA Encounter object has an SDA `CustomPairs` collection with three members, each with the name `PlanOfCareInstructionsText`:

```
{
  "resourceType": "Encounter",
  "contained": [
    {
      "resourceType": "Parameters",
      "id": "63",
      "parameter": [
        {
          "name": "PlanOfCareInstructionsText",
          "valueString": "Doctor recommends at least 30 minutes of exercise per day"
        },
        {
          "name": "PlanOfCareInstructionsText",
          "valueString": "Use sports heart rate monitor to aid in monitoring effort level"
        },
        {
          "name": "PlanOfCareInstructionsText",
          "valueString": "Read \"South Beach Diet\""
        }
      ]
    }
  ],
  "extension": [
    {
      "url": "http://intersystems.com/fhir/extn/sda3/lib/encounter-custom-pairs",
      "valueReference": {
        "reference": "#63"
      }
    }
  ]
}
```

```
    }  
  ],  
  "id": "914"  
}
```

## 11.4 Customizing Transformations

Each SDA-FHIR transformation uses a Data Transformation Language (DTL) class to map SDA objects to FHIR resources, and vice versa. You can [customize these DTLs](#) using the DTL Editor.

If you want to implement more advanced custom transformation behavior, you can subclass the appropriate transformation API class and override its methods. For more information, see [Customizing Transformation APIs](#). For information about upgrading to the new customization architecture from a legacy FHIR implementation, see [Upgrading Pre-2020.2 Transformations](#).

InterSystems products also provide a mechanism for [customizing lookup tables](#) used by the transformations.

You customize a transformation within a specific namespace, not for the entire instance, so you can have different customizations in each namespace. If you want multiple namespaces to have the same customized transformations, you must repeat the customization process for each namespace.

### 11.4.1 Implementing Custom DTLs

The strategy for customizing a DTL that the transformation uses to convert SDA to FHIR (and vice-versa) involves creating a copy of the standard DTL and then modifying it. After you manually specify the package of custom DTL, the transformation will automatically select the custom DTL instead of the standard one.

#### 11.4.1.1 Specifying a Package for Custom DTLs

Before customizing DTLs, you need to specify a single package for all customized DTL classes. InterSystems recommends naming the class package: `HS.Local.FHIR.DTL`. Once you have decided on the package that will be used for all custom DTLs, you need to use the InterSystems Terminal to specify this package. To specify the custom DTL package:

1. Open the InterSystems Terminal.
2. Change to namespace that contains the SDA-FHIR transformations. For example:

```
set $namespace = "fhirnamespace"
```

3. To check if a custom DTL package already exists, enter:

```
Write ##class(HS.FHIR.DTL.Util.API.ExecDefinition).GetCustomDTLPackage()
```

4. If the custom DTL package does not already exist, enter the following command, replacing `HS.Local.FHIR.DTL` with the name of your custom DTL package:

```
set status = ##class(HS.FHIR.DTL.Util.API.ExecDefinition).SetCustomDTLPackage("HS.Local.FHIR.DTL")
```

5. To check that the package was defined successfully, enter:

```
write status
```

The response should be: 1.

### 11.4.1.2 Creating the Custom DTL

You create a custom DTL by saving a copy of the existing standard DTL and then editing it. The package and name of the custom DTL must conform to naming standards so the transformation knows to use the custom DTL rather than the standard one. To create a custom DTL:

1. Open the Management Portal and navigate to the FHIR namespace.
2. Select **Interoperability > List > Data Transformations**.
3. Find the name of the transformation that you want to customize. For example, transformations from SDA to FHIR STU3 are prefixed with `HS.FHIR.DTL.SDA3.vSTU3` while transformations from FHIR STU3 to SDA are prefixed with `HS.FHIR.DTL.vSTU3.SDA3`.
4. Double-click the name of the transformation you want to customize to open it in the DTL Editor.
5. Open the InterSystems Terminal.
6. To obtain the required name for the customized DTL class, enter the following in the Terminal:

```
Write ##class(HS.FHIR.DTL.Util.API.ExecDefinition).PreviewDTLCustomClass("<standard_class_name>")
```

Where `<standard_class_name>` is the full name of the transformation that you are customizing, including packages. It is the name of the transformation that you have open in the DTL Editor. You can view the name on the **Transform** tab, but do not include the `.dtl` extension.

7. Be sure to make note of the response in the Terminal. You need to give your customized DTL class this name.
8. In the DTL Editor, click **Save As**.
9. In the **Package** field, enter the package from the name of the customized DTL class that appeared in the Terminal. For example, if the customized class name in the Terminal was `HS.Local.FHIR.DTL.SDA3.vSTU3.Address.Address`, then enter `HS.Local.FHIR.DTL.SDA3.vSTU3.Address` (without the actual class name).
10. In the **Name** field, enter the name of the customized class. For example, if the customized class name in the Terminal was `HS.Local.FHIR.DTL.SDA3.vSTU3.Address.Address`, then enter `Address`.
11. Enter a description and click **OK**.

### 11.4.1.3 Copying Custom Class to Mirror Members

If your environment uses mirroring and the package of your customizations resides in a non-mirrored database, you must copy the customized DTL class to the custom package on each mirror member. For example, if you defined the package for customized classes as `HS.Local.FHIR.DTL`, then you must copy the customized DTL class to `HS.Local.FHIR.DTL` on each mirror member because `HS.Local` resides in the `HSCUSTOM` namespace, which is not mirrored. If your custom package resides in a mirrored database, no further action is required.

## 11.4.2 Customizing Transformation API Classes

The transformation API classes contain several methods that can be overridden to implement custom transformation behavior. To override a method, subclass `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR` or `HS.FHIR.DTL.Util.API.Transform.FHIRToSDA3` and write your custom method. For example, if you want to select a DTL based on a condition, you can override the `GetDTL` method. The following is a brief introduction to the overridable transformation methods.

### 11.4.2.1 SDA to FHIR Overridable Methods

The following methods of the `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR` class can be overridden to implement custom transformation behavior.

**GetDTL**

Specifies the DTL class used to transform a given SDA object. You do not need to override this method to [use a custom DTL](#); if you specified a custom DTL package, the `GetDTL` method finds the custom DTL before using the standard one. However, you can override this method if you want to select a DTL from multiple possibilities based on a condition.

**IsDuplicate**

Override this method to change how the transformation checks whether a generated resource that is referenced by another resource in the bundle already exists. For example, you might want to relax what is needed to identify a shared resource like Organization, Practitioner, or Medication as a duplicate. By default, the first 32 kilobytes of a shared resource are added as a hash in a hash table. For each subsequent reference to a shared resource, the transformation determines whether the referenced resource is a duplicate by searching the hash table for a direct match of the JSON.

If the `IsDuplicate` method determines that a referenced resource already exists, it is not included in the bundle output.

**ResourceLookup**

By default, only the bundle created by the transformation is searched for a specified resource when the `ResourceLookup` method is called. However, you can override this method, for example, if you want the application to search for the specified resource in a repository as well as in the bundle output.

**GetReference**

When transforming SDA that has a reference to another streamlet, this method returns the reference to the FHIR resource that is created for the referenced SDA object. For example, when an `EncounterNumber` is passed to this method, it returns a reference to the FHIR Encounter resource that corresponds to the SDA Encounter that was referenced by the specified `EncounterNumber`. Override the method to generate a custom reference to the specified FHIR resource.

**GetId**

By default, an individual resource is not assigned an `id` when the transformation produces a bundle. Override the `GetId` method to assign resources in the bundle an `id`. In this case, the value for the `fullUrl` field of the bundle is *baseUrl/resourceType/id* and the resource references in the bundle are *resourceType/id*.

**GetBaseURL**

Override the `GetBaseURL` method to change the URL prefix of each resource. For example, if you are posting FHIR resources to a specific repository, you can provide a URL prefix that identifies repository.

**HandleInvalidResource**

The transformation validates each resource before adding it to the Bundle output. Override the `HandleInvalidResource` method to customize what happens to a resource that fails validation. By default, an error is thrown and processing stops, which means the Bundle is not returned.

### 11.4.2.2 FHIR to SDA Overridable Methods

The following methods of the `HS.FHIR.DTL.Util.API.Transform.FHIRTToSDA3` class can be overridden to implement custom transformation behavior.

**GetDTL**

Specifies the DTL class used to transform a given FHIR resource. You do not need to override this method to [use a custom DTL](#); if you specified a custom DTL package, the `GetDTL` method finds the custom DTL before using the standard one. However, you can override this method if you want to select a DTL from multiple possibilities based on a condition.

**GetResourceFromReference**

This method controls where the transformation looks for a resource that has been referenced by another resource in the bundle. For example, you could override the method to find the referenced resource in a repository rather than in the same bundle.

**GetSendingFacility**

Override this method to customize how the value of the SDA `SendingFacility` property is set.

By default, the `SendingFacility` property is set as follows: if the Patient's `managingOrganization` field contains a reference to an Organization, and that Organization is in the Bundle, it is used. Otherwise, the patient identifiers are searched for an MRN with an assigning authority, and that assigning authority is used. If neither of these items is found, the string `FHIR` is used.

**GetIdentifier**

Override this method to customize how certain identifiers are assigned to SDA properties.

For example, this method can be customized to assign values to the `EncounterNumber` field of an Encounter streamlet. In this case it can be useful to access the contents of the resource being converted in order to determine what `EncounterNumber` should be returned. The instance property `%currentReference` contains a FHIR reference object that can be passed into the instance method `GetResourceFromReference` in order to obtain the resource as a dynamic object. By default, the value of the `EncounterNumber` properties are assigned sequentially, starting at 1.

Overriding this method can also be useful for assigning the `ExternalID` value for the SDA `HealthConcern` or `Goal`. By default, the value of `ExternalID` properties are assigned sequentially, starting at 1.

**HandleMissingResource**

By default, if a resource that is referenced by another resource within the incoming FHIR bundle is not present in the bundle, the transformation of the bundles continues. To change what happens when there is a missing resource in the bundle, override the `HandleMissingResource` method.

## 11.4.3 Customizing Lookup Tables

The [FHIR Annotations tool](#) allows you to explore the lookup tables that are used by transformations to map codes in the source data format to codes in the target format. You can customize these lookup tables by using a InterSystems Terminal utility or by manually modifying a JSON file that contains the lookup tables.

### 11.4.3.1 Using the Terminal Utility to Customize a Lookup Table

InterSystems provides a Terminal utility that leads you through the process of customizing a lookup table. To run the customization utility:

1. Open the InterSystems Terminal.
2. To change to the FHIR namespace, enter:

```
set $namespace = "fhirnamespace"
```



where `fhirnamespace` is the FHIR namespace you have created.

3. To start the utility, enter:

```
do ##class(HS.FHIR.DTL.Util.API.LookupTable).EditLookupTable()
```

4. Enter the Mapping Source for the lookup table you are customizing. For example, if you are customizing a lookup table that maps values from SDA3 to STU3, enter `SDA3`.
5. Enter the Mapping Target for the lookup table you are customizing. For example, if you are customizing a lookup table that maps values from SDA3 to STU3, enter `STU3`.
6. Enter the number that corresponds to Mapping Source Value Set in the lookup table you want to customize.
7. If only one lookup table with the Mapping Source Value Set exists, the Mapping Target Value Set is selected automatically and you can skip to the next step. If not, enter the number that corresponds to the Mapping Target Value Set you want to customize.
8. Select the code-to-code mapping you want to edit. If you want to add a new code-to-code mapping in the lookup table, enter `+`.
9. If you are editing the target value of a code-to-code mapping, enter the new target value for the mapping.

If you want to edit the source value of the code-to-code mapping, you must enter `-` to delete the entire code-to-code mapping, then re-run the utility to add a new mapping with the correct source and target values.

### 11.4.3.2 Editing Lookup.json to Customize a Lookup Table

Rather than using the Terminal utility, you can customize lookup tables by adding, deleting, or editing key/value pairs in a JSON file that contains all of the lookup tables used by transformations. Before beginning, you must make a custom copy of this JSON file, `Lookup.json`, and put it into a special directory.

#### Creating Custom Lookup.json File

To create a custom JSON file that will be used by transformations when accessing lookup tables:

1. Navigate to `<install-dir>\dev\fhir\lookup`, where *<install-dir>* is the directory where you installed your InterSystems product.
2. If it does not already exist, create a new directory called `custom`.
3. Navigate to the `custom` directory.
4. If it does not already exist, create a new directory that is the name of your FHIR namespace in all capital letters. For example, if the namespace that contains your FHIR STU3 production is called `fhirnamespace`, create a directory called `FHIRNAMESPACE`.
5. Copy `Lookup.json` from the `<install-dir>\dev\fhir\lookup` directory to the new `<install-dir>\dev\fhir\lookup\custom\<NAMESPACE>` directory.

You can now begin to edit the lookup tables in the new copy of `Lookup.json`.

#### Editing Custom Lookup.json File

To begin customizing a lookup table, you must gather four pieces of information:

- Mapping Source
- Mapping Target
- Mapping Source Value Set
- Mapping Target Value Set



These values can be found in the FHIR Annotations in the Management Portal. To access these values:

1. Open the Management Portal and navigate to your FHIR namespace.
2. From the Home page, select **Health > Schema Documentation > FHIR Annotations**.
3. In the first drop-down list, select the type of transformation that contains the lookup table you are customizing. For example, if you are interested in how SDA3 and FHIR STU3 codes map to each other in a lookup table, select **FHIR3→SDA3**.

Make note of the Mapping Source and Mapping Target. The first interface format in the transformation pair is the Mapping Source. The second interface format is the Mapping Target. For example, if you select **FHIR3→SDA3**, vSTU3 is the Mapping Source and SDA3 is the Mapping Target.

4. Click the **View <transformation> Lookup Tables** button, where the full name of the button depends on which transformation pair you selected.
5. Using the **View Lookup Tables** dialog, use the drop-down lists to note the Mapping Source Value Set and Mapping Target Value Set. The Mapping Source Value Set is the name in the left-hand drop-down list. The Mapping Target Value Set is the name in the right-hand drop-down list.

Now that you have the Mapping Source, Mapping Target, Mapping Source Value Set, and Mapping Target Value Set, you can edit a lookup table by adding, deleting, or editing the appropriate key/value pair in the custom Lookup.json file.

The top-level key/value pair in Lookup.json corresponds to the Mapping Source to Mapping Target relationship. For example, a lookup table used by SDA3 to FHIR STU3 transformations looks like:

```
"SDA3" : {
  "vSTU3" : {
```

The next level of key/value pairs corresponds to the Mapping Source Value Set to the Mapping Target Value Set. Search for the correct lookup table by finding the corresponding key/value pair. For example:

```
"HS.SDA3.Alert:Status" :
  {"event-status" : {
```

Once you have located the lookup table, you can add, delete, or edit the key/value pairs that correspond to the code-to-code mappings.

```
"A": "in-progress",
"C": "unknown",
"I": "aborted",
"INT": "completed"
```

## Loading Custom Lookup.json File

Once you have customized Lookup.json, you need to load it using the Terminal before it can be used by the SDA-FHIR transformations. To load the JSON file:

1. Open the Terminal.
2. Change to your FHIR namespace. For example:

```
set $namespace = "fhirnamespace"
```

3. Run the following command:

```
set status = ##class(HS.FHIR.DTL.Util.API.LookupTable).ImportLookupJSONToGlobal()
```



# 12

## ObjectScript Applications

When using a server-side application to make FHIR requests to the internal FHIR server, your application should use the standard FHIR client. For details about using these built-in classes, see [FHIR client](#).

However there are some cases where you might want to perform actions on the server's storage without a request. For example, you might want to perform a write operation even though the server's metadata restricts requests to read-only interactions. In this case, you can [bypass the Service](#).

Your ObjectScript application can also [validate a resource](#).

### 12.1 Bypassing the Service

A server-side application can call the methods of an [Interactions](#) subclass directly instead of submitting programmatic requests via the Service. For example, an application could call the Interactions subclass' Add method directly rather than sending a POST request to the Service. This is especially useful if the server-side application needs to perform actions that are prohibited by the Service. For example, if the server's metadata configures the endpoint as read-only, programmatic requests to the Service cannot create resources. However, using method calls to the Interactions subclass, a server-side application could update the storage strategy with resources, effectively bypassing the restrictions enforced by the Service.

Programmatic calls to methods of the Interactions class pass FHIR data as dynamic objects. For more information about working with this data, see [FHIR Data and Dynamic Objects](#).

### 12.2 Validating FHIR Resources

Your ObjectScript application can programmatically validate a resource against the FHIR server's metadata without using the FHIR \$validate operation as long as the resource is represented as a dynamic object. For example, the following code validates a Patient resource against the server's FHIR Release 4 metadata, which includes the schema for the Patient resource. When calling the LoadSchema method, you can specify the common name of the FHIR version (for example, R4 or STU3) or the name of the server's base metadata (for example, HL7v40 or HL7v30).

```
// Put JSON representation of Patient resource into a dynamic object
set patient = ##class(%DynamicObject).%FromJSON("c:\localdata\myPatient.json")

//Validate the patient resource
set schema = ##class(HS.FHIRServer.Schema).LoadSchema("R4")
set resourceValidator = ##class(HS.FHIRServer.Util.ResourceValidator).%New(schema)

do resourceValidator.ValidateResource(patient)
```



# 13

## Legacy FHIR Technology

For details about using legacy FHIR technology, see the legacy FHIR books that are available at [InterSystems Legacy Documentation](#).

### 13.1 Upgrading Legacy Transformations

The strategy for customizing bi-directional SDA-FHIR transformations in InterSystems products was different in the legacy FHIR technology (pre-2020.2). This section discusses how to convert code developed to customize transformation in legacy FHIR implementations to the new FHIR architecture.

The APIs called by an application to perform transformations have changed. In the legacy implementation, applications called methods of the `HS.FHIR.DTL.Util.API.HC.Transform` class to invoke the transformation. This class is obsolete and direct calls to its methods will not work with the new FHIR architecture. Now, transformations are invoked with methods of the [HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR](#) and [HS.FHIR.DTL.Util.API.Transform.FHIRToSDA3](#) classes.

The legacy FHIR technology used callback objects to implement custom logic controlling how transformations were executed. In the new architecture, customization is accomplished by subclassing the transformation API class and overriding its methods. For information about customizing these transformation methods, see [Customizing Transformation API Classes](#).

When upgrading from your legacy callback classes, you need to migrate the logic in your callback methods to the overridable methods in the new transformation classes. The following table summarizes the relationship between callback methods in the legacy `HS.FHIR.DTL.Util.API.HC.Callback.Default.SDA3ToSTU3` class and new overridable methods in `HS.FHIR.DTL.Util.API.Transform.SDA3ToFHIR`.

Legacy Callback Method	New Overridable Method
<code>IsDuplicate</code>	<code>IsDuplicate</code>
<code>AssignResourceId</code>	<code>GetId</code>
<code>GetIdByIdentifier</code>	<code>GetId</code>
<code>GetPatientId</code>	<code>GetId</code>
<code>GetURLPrefix</code>	<code>GetBaseURL</code>

The following table summarizes the relationship between callback methods in the legacy `HS.FHIR.DTL.Util.API.HC.Callback.Default.STU3ToSDA3` class and new overridable methods in `HS.FHIR.DTL.Util.API.Transform.FHIRToSDA3`.

Legacy Callback Method	New Overridable Method
AssignEncounterNumber	GetIdentifier
AssignExternalId	GetIdentifier
GetSendingFacility	GetSendingFacility
GetSendingFacilityFromReference	GetSendingFacility

One of the methods in the new transformation classes, `GetDTL`, can be overridden to select a custom DTL class that was written for the legacy FHIR technology. In this case, the `GetDTL` method should call the old method `GetDTLPackageAndClass`. For example:

```
Method GetDTL(source As HS.SDA3.DataType, DTL As %Dictionary.Classname = "") As %Dictionary.Classname
{
    // Get the standard product DTL class name for this SDA3 data type.
    Set className = ##super(source, DTL)

    Set className = ##class(HS.FHIR.DTL.Util.API.ExecDefinition).GetDTLPackageAndClass(className)
    Quit className
}
```

### 13.1.1 Upgrading Transformation Productions

The business processes used to perform transformations in a FHIR interoperability production, `HS.FHIR.DTL.Util.HC.SDA3.FHIR.Process` and `HS.FHIR.DTL.Util.HC.FHIR.SDA3.Process`, have been updated to use the new transformation API. If your legacy implementation used the standard business processes, you must complete the following tasks before starting the production after the upgrade:

- Specify a value for the **FHIRMetadatSet** setting of the business process.
- If the **TransmissionMode** setting was set to **Batch**, you must change the setting to specify **transaction** or **individual**. `HS.FHIRServer.Interop.Operation`