FHIR Intermediate Course, Unit 3

FHIR SERVERS

Reading Material

FHIR Servers

Introduction

In this unit we will focus on how to create a FHIR server.

The objective is that by the end of this unit you will have an idea of what is involved in creating a FHIR server and strategies to include one in your architecture.

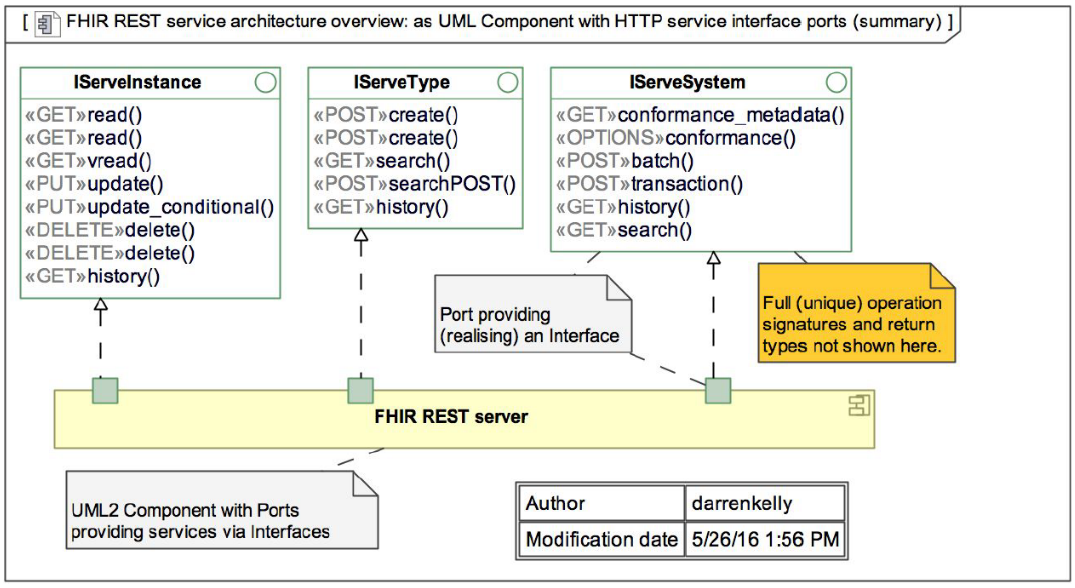
Although many people have been working with FHIR for a long time, we know from experience that many times not everyone is aware of what functionalities a FHIR server should or can have and what your obligations are as the creator of a server, so we will review them.

First of all ... what is considered to be a FHIR server?

Simply put, a FHIR server is any software that implements the FHIR APIs and uses FHIR resources to exchange data.

The following diagram describes the definitions of the FHIR interface. The methods are classified as:

- iServeInstance: methods that perform Get, Put or Delete operations on a resource
- iServeType: methods that obtain type information or metadata about resources
- iServeSystem: methods that are exposed or functionalities enabled.



Using this as a starting point, let's look at the functionalities to consider.

The minimum that your FHIR server should do is expose what capabilities it has, what functionalities it is prepared to respond to.

Capabilities of a FHIR server

As we know, we obtain the functionalities of a server by performing an HTTP GET with the following structure on the base server.

GET [base] / metadata {? Mode = [mode]} {& _format = [mime-type]}

The FHIR server should return a resource that describes the capacity of the server, and the information returned will depend on the value of the "mode" parameter.

Let's look at the different modes:

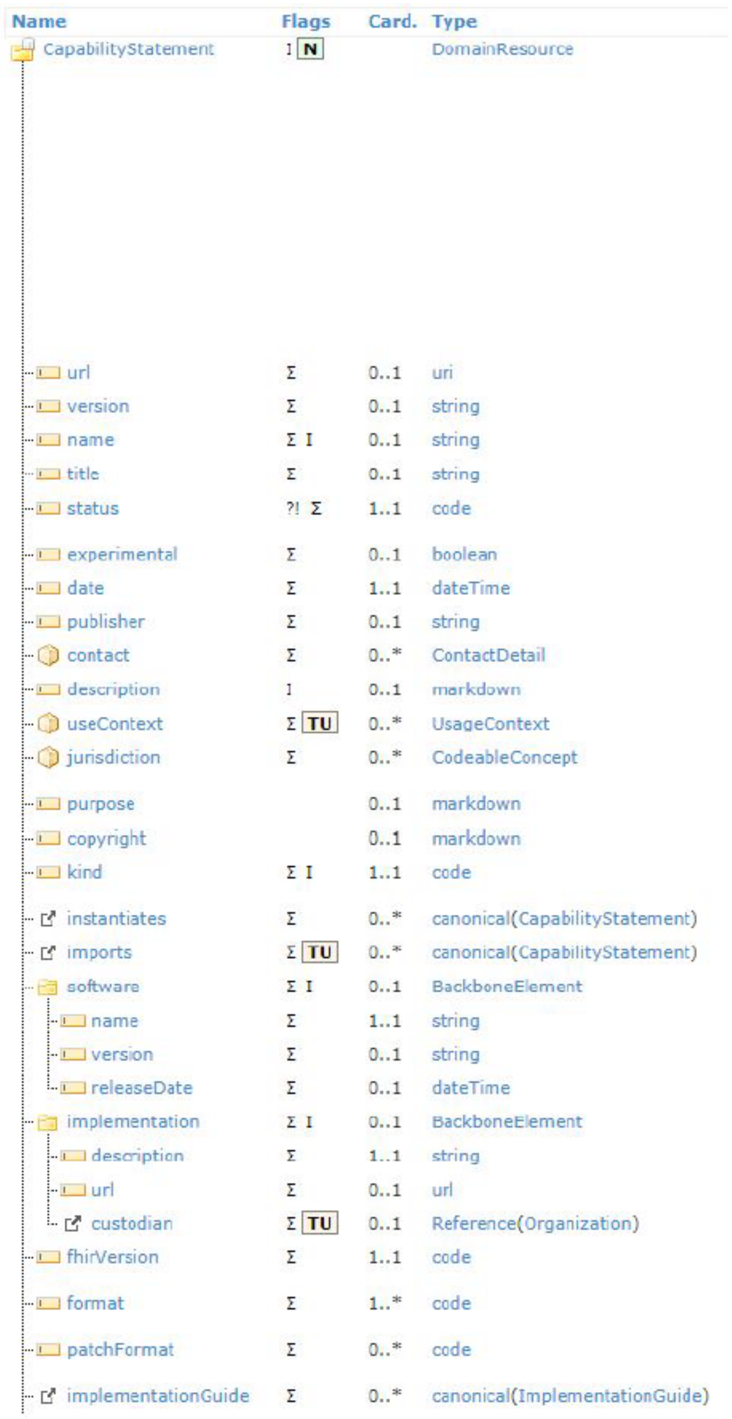
- **full** (or unspecified mode): Will return a Capability Statement resource with the supported resource types and interactions. (The most widely used)
- **Normative:** Like full but only with the normative part of the CapabilityStatement
- **Terminology:** It will return a TerminologyCapabilities resource with information about terminology services provided/supported.

Your server could ignore the mode parameter and return the CapabilityStatement resource directly. The resource returned generally has an arbitrary ID and no metadata (although it is not prohibited).

Capacity statements can be quite long. It is recommended that they admit the _summary and _elements parameters and that they implement the $subset and $implement operations to facilitate compliance verification.

To analyze the capabilities that your server may have, we will look at the CapabilityStatement resource.

CapabilityStatement resource header



Name	Flags	Card.	Type
CapabilityStatement	I N		DomainResource
url	Σ	0..1	uri
version	Σ	0..1	string
name	Σ I	0..1	string
title	Σ	0..1	string
status	?! Σ	1..1	code
experimental	Σ	0..1	boolean
date	Σ	1..1	dateTime
publisher	Σ	0..1	string
contact	Σ	0..*	ContactDetail
description	I	0..1	markdown
useContext	Σ TU	0..*	UsageContext
jurisdiction	Σ	0..*	CodeableConcept
purpose		0..1	markdown
copyright		0..1	markdown
kind	Σ I	1..1	code
instantiates	Σ	0..*	canonical(CapabilityStatement)
imports	Σ TU	0..*	canonical(CapabilityStatement)
software	Σ I	0..1	BackboneElement
name	Σ	1..1	string
version	Σ	0..1	string
releaseDate	Σ	0..1	dateTime
implementation	Σ I	0..1	BackboneElement
description	Σ	1..1	string
url	Σ	0..1	url
custodian	Σ TU	0..1	Reference(Organization)
fhirVersion	Σ	1..1	code
format	Σ	1..*	code
patchFormat	Σ	0..*	code
implementationGuide	Σ	0..*	canonical(ImplementationGuide)

Figure 1 - CapabilityStatement resource header

In this first part we see the actual resource header.

Some recommendations:

+ Warning: Name should be usable as an identifier for the module by machine processing applications such as code generation

+ Rule: A Capability Statement SHALL have at least one of REST, messaging or document element.

+ Rule: A Capability Statement SHALL have at least one of description, software, or implementation element.

+ Rule: Messaging end-point is required (and is only permitted) when a statement is for an implementation.

+ Rule: The set of documents must be unique by the combination of profile and mode.

+ Rule: If kind = instance, implementation must be present and software may be present

+ Rule: If kind = capability, implementation must be absent, software must be present

+ Rule: If kind = requirements, implementation and software must be absent

As for the elements of the header, many of them are self-explanatory, so we will focus on the most used elements and those that are mandatory.

The **status** should detail the status of this capacity statement and allows us to track the life cycle of the content.

Possible values are: draft | active | retired | unknown

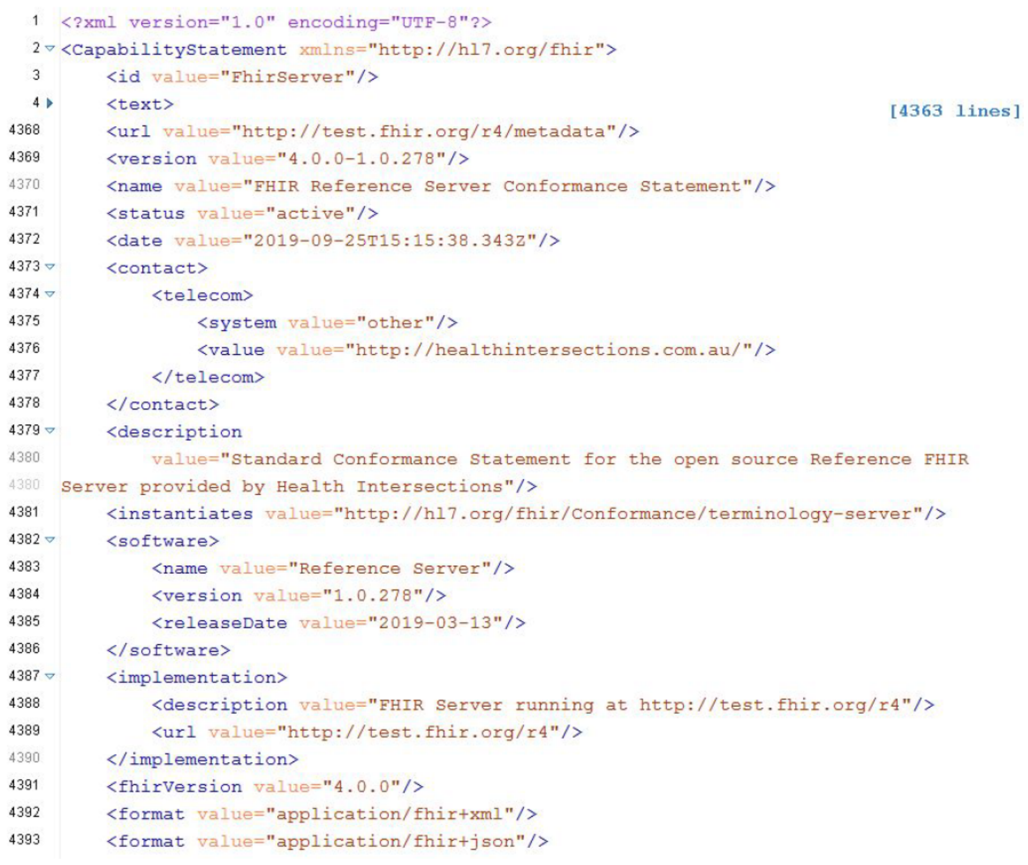
Date: Will show the date last modified.

Kind: The way that this statement is intended to be used, to describe an actual running instance of software, a particular product (kind, not instance of software) or a class of implementation (e.g. a desired purchase).

fhirVersion: Specifies the version of FHIR that it supports and format the supported formats.

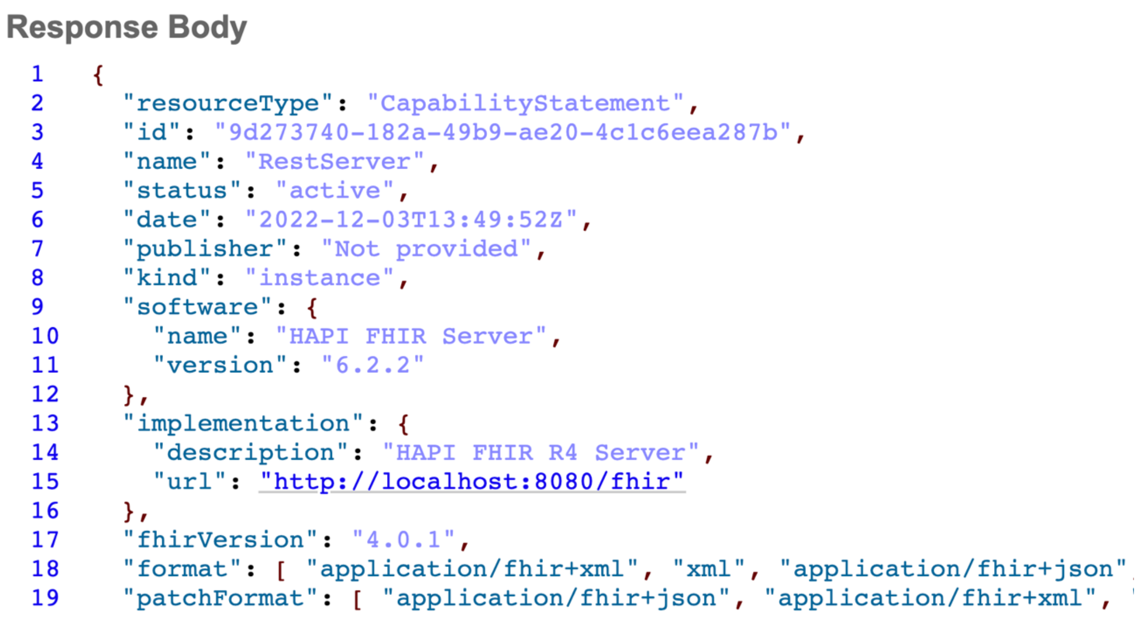
Here is an example of the CapabilityStatement for the HL7 test server. (We're omitting 4,363 lines of the text element that restates in a more user-friendly way what is indicated in the resource).

This is for comparison with our server:



```
<?xml version="1.0" encoding="UTF-8"?>
<CapabilityStatement xmlns="http://hl7.org/fhir">
    <id value="FhirServer"/>
    <text>                                                    [4363 lines]
    <url value="http://test.fhir.org/r4/metadata"/>
    <version value="4.0.0-1.0.278"/>
    <name value="FHIR Reference Server Conformance Statement"/>
    <status value="active"/>
    <date value="2019-09-25T15:15:38.343Z"/>
    <contact>
        <telecom>
            <system value="other"/>
            <value value="http://healthintersections.com.au/"/>
        </telecom>
    </contact>
    <description
        value="Standard Conformance Statement for the open source Reference FHIR
Server provided by Health Intersections"/>
    <instantiates value="http://hl7.org/fhir/Conformance/terminology-server"/>
    <software>
        <name value="Reference Server"/>
        <version value="1.0.278"/>
        <releaseDate value="2019-03-13"/>
    </software>
    <implementation>
        <description value="FHIR Server running at http://test.fhir.org/r4"/>
        <url value="http://test.fhir.org/r4"/>
    </implementation>
    <fhirVersion value="4.0.0"/>
    <format value="application/fhir+xml"/>
    <format value="application/fhir+json"/>
```

Figure 2 - CapabilityStatement header for the HL7 server



Response Body

```
{
  "resourceType": "CapabilityStatement",
  "id": "9d273740-182a-49b9-ae20-4c1c6eea287b",
  "name": "RestServer",
  "status": "active",
  "date": "2022-12-03T13:49:52Z",
  "publisher": "Not provided",
  "kind": "instance",
  "software": {
    "name": "HAPI FHIR Server",
    "version": "6.2.2"
  },
  "implementation": {
    "description": "HAPI FHIR R4 Server",
    "url": "http://localhost:8080/fhir"
  },
  "fhirVersion": "4.0.1",
  "format": [ "application/fhir+xml", "xml", "application/fhir+json"
  "patchFormat": [ "application/fhir+json", "application/fhir+xml",
```

Figure 3 - CapabilityStatement header for the course server

Detail of the CapabilityStatement resource

We have 3 main branches:

- rest
- messaging
- document

We will focus on the **REST** branch, since right now we will not work with messaging or documents, but as a general overview, in **messaging**, we should specify the ability to send or receive messages, what the endpoints are for sending or receiving messages and which protocol is supported by that endpoint (for example HTTP, FTP or MLLP), the documentation of that messaging and details about the messages supported.

In **document** we should specify the mode in which it is compatible with documents (whether as a generator or consumer) and the supported profiles.

Before starting, we should remember that if our server supports more things than those included in the standard, they must be defined as extensions and by rule, for any element of the standard, ALWAYS must be included before any of the normative elements.

If you read through the CapabilityStatement of the HL7 test server, you will see that it includes extensions of many types and in different elements.

Focusing, then, on **REST**, the first mandatory element is **mode**, where we specify how the endpoint acts, whether as a client or as a server http://hl7.org/fhir/valueset-restful-capability-mode.html and then in **security**, the security scheme supported by the server – for example, OAuth, SMART on FHIR, NTLM, Basic, Kerberos or Certificates.

After this, we should specify the list of resources, which obviously should detail for each one the type of resource supported, the profile it is compliant with, the interactions supported (read | vread | update | patch | delete | history- instance | history-type | create | search-type) and for search, the supported search parameters, and if we support the inclusion of other resources in the search response (_include and _revinclude parameters). For example, we are asked to include the Patient and Practitioner resources in a search for an Observation resource.

Similarly, we should specify whether or not we support resource versioning (no-version | versioned | versioned-update) and whether history reading, conditional creation, conditional update or conditional deletion are tolerated.

Finally, at the resource level, we should specify the operations we support on that resource. Are we going to support validating resources?

Server Behavior

As we can see, there are many elements to consider even before looking at the behavior our server should have, which is found in the specification for the RESTful API.

The FHIR specification is based on what the industry already uses in terms of REST services. It only uses level 2 of the REST maturity model for the standard specification, although with extensions it achieves compliance with level 3.

Each "type of resource" has the same set of defined interactions that can be used to manage resources in a highly granular way.

Note that in this RESTful scheme, transactions are made directly on the server using an HTTP request/response. The API does not deal directly with authentication, authorization or auditing. All interactions are described for synchronous use (although an asynchronous usage pattern is also defined.)

Your server can choose which of these interactions are available and what types of resources it supports.

We are just going to list the operations, but we assume that you know what they imply:

Instance Level Interactions

- **read** *Read the current state of the resource*
- **vread** *Read the state of a specific version of the resource*
- **update** *Update an existing resource by its id (or create it if it is new)*
- **patch** *Update an existing resource by posting a set of changes to it*
- **delete** *Delete a resource*
- **history** *Retrieve the change history for a particular resource*

Type Level Interactions

- **create** *Create a new resource with a server assigned id*
- **search** *Search the resource type based on some filter criteria*
- **history** *Retrieve the change history for a particular resource type Whole System Interactions*
- **capabilities** *Get a capability statement for the system*
- **batch/transaction** *Update, create or delete a set of resources in a single interaction*
- **history** *Retrieve the change history for all resources*
- **search** *Search across all resource types based on some filter criteria*

HTTP status codes

FHIR sets rules for the use of HTTP status codes for each of the interactions listed and for each case. We will not go into the details for each one since they are clearly specified in the standard.

But it is important to consider the use of the OperationOutcome resource to transmit detailed information on any processable error. Some combinations of interactions and specific response codes even require that OperationOutcome be returned as the content of the response. In general its use is recommended with any HTTP 4xx or 5xx response, although it is not mandatory.

Finally, another important thing to know is whether your server is going to support batch operations or transactions. http://hl7.org/fhir/http.html#transaction

That's really a topic for advanced servers, but the main thing to know is that generating a FHIR server from the ground up is not an easy or simple task.

Now that you have an idea of everything your server can and should do, we will focus on how to incorporate a FHIR server into your architecture.

Incorporating a FHIR server

When incorporating a FHIR server into our architecture, we have two very different options, and which one we choose will depend on the business unit that we manage, the amount of FHIR resources we want to work with, and the scenario in which we implement it.

Generic server

These servers generally support all interactions and all standard resources, and are derivatives of the reference implementations.

They generally include their own storage from which the services of the different resources are provided.

The creation of generic FHIR servers is out of scope for this course. But do not worry, several vendors (including Microsoft, Google and IBM) and open source projects provide 'out-of-the-box' or cloud based FHIR servers. Obviously, the maturity and feature list is diverse.

If you want to install a generic server, in the following link you will find useful information on how to do it. HAPI FHIR http://hapifhir.io/doc_jpa.html

Specific servers

These servers generally implement only part of the standard. They do not have a data replication scheme; instead, they usually are connected to a backend that has all the information in the information system, such as an electronic medical record or an appointment scheduling system, and they usually have their own domain operations that meet specific needs. For the most part, these servers end up meeting the need by exposing a FHIR facade to the system.

FHIR Facades

If you read the Argonaut Implementation Guide, this is what it enables: a common new, standardized API for all the “medical software” already in place.

Vendors and Providers are not mandated to REPLACE their existing software, but to FHIR-enable their existing software.

And this is precisely what we want to discuss through this unit.

We will talk about

- where to “put” your façade in terms of system architecture
- what can the tools for creating FHIR facades do for you
- what is expected from you

Where to put your FHIR façade

To help in this discussion, we recommend reading these articles located in the Health Samurai blog, author of several tools for creating FHIR servers and ecosystems.

One discusses general concepts about FHIR servers, and the other specific options on where to locate a FHIR façade.

https://www.health-samurai.io/articles/the-fhir-guide-for-ctos-and-technical-leaders

https://www.health-samurai.io/articles/about-fhir-facades-part-i-two-approaches

Based on our own experience, we classified three different ways of FHIR-enabling your "legacy app"

Discussion 1 : What is legacy? Anything not "completely FHIR-based"? What will be legacy tomorrow? Anything programmed today. And we enable FHIR for 'legacy' systems because they will continue being used for some time (1, 2, 5, 10 years, who knows?) so our architecture for FHIR enabling our app cannot "kill" it: compromise its security, performance, scalability or flexibility. So we need to be careful, even when thinking "this will be temporary until we move everything to…". "Temporary" in healthcare informatics can be measured in decades.

Discussion 2: This approach does not mean that you need "only" to create the technical façade, because sometimes there is some information mandatory for your implementation guide, that is not recorded in your system. This is usually part of the process of creating the façade, structural changes (in all layers) in your current system are not discarded.

Anyway we believe that the FHIR façade is a big leap forward from silos, and sometimes it's the step needed to increase true interoperability without rewriting all our existing software.

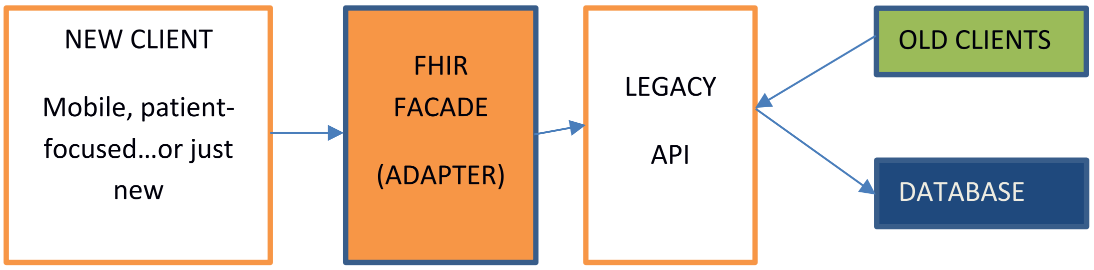
Our three options are:

1- A FHIR Façade for your existing API
2- A FHIR Façade for your database
3- A FHIR Façade for your data, using a FHIR native server

Which option to use and when, is really up to you. Sometimes you are forced to use some of the options, some other times you need to mix & match more than one, and these decisions depends on the design of the legacy system, the skills of the teams involved, and more political or financial conditions (how much can "they" change, how much can "they" do? Which kind of access will we obtain?). Depending on the use case, access can be read-only, or read-write. Most facades in place now are read-only. Are these the "only" options available?. By no means. Just what we used and observed so far.

Option 1: A FHIR façade for your existing API

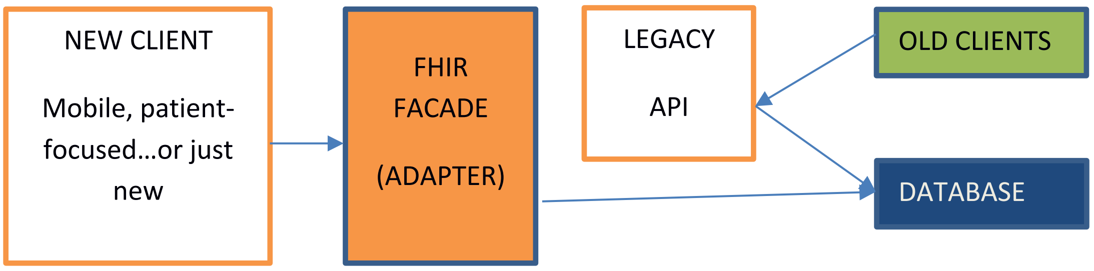
We will discuss this option in depth for all our **projects.** There is a system in place, with an existing API, and the only way to access the information in this system is through the existing API (hopefully it's a **RESTful** proprietary API, with its own resource model, but it can be anything)



The adapter should understand both worlds (the FHIR world, and the legacy world) in every sense: vocabulary, data elements, workflow, transport, etc.

Option 2: A FHIR façade for your existing DATABASE

We will not discuss this option in depth. But in our assignment, there is a database with a very thin API, and we build the façade for this API. There is a system in place, with a database design you cannot change, but you can access the database using methods designed by the DBAs or the system vendor, and that's the only way to access the information in this system.



Option 3 -A FHIR Façade for your data, using a FHIR native server

Let’s imagine the scenario of a laboratory system that handles only a few resources, maybe about 10, and the functionality it needs to offer is: “after finalizing the results for a lab order, it publishes them on a server with a standard FHIR interface”. This allows all its clients, such as EHRs, personal health portals and other laboratories, to access its reports in a single, standardized way.

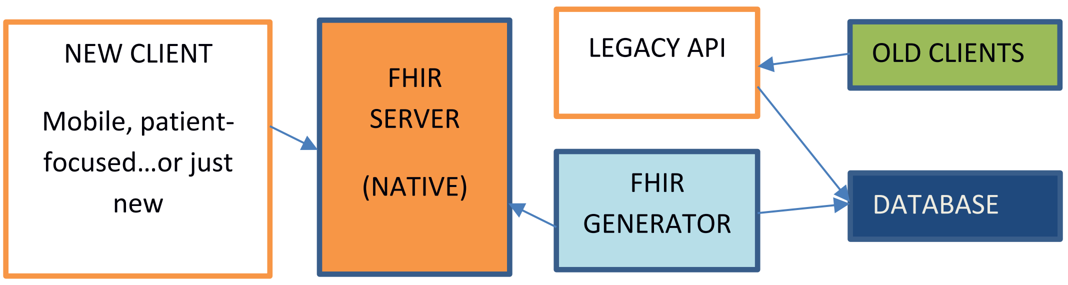
One strategy would be to install a generic server and from your existing application, from your current system, every time a result is finalized, you post it to that server.

It seems much simpler to implement a POST to a server than to develop a FHIR server.

In this data replication model, a generic server would meet the need.

In this scheme, both the consumer and the generator of the information are FHIR clients.

But this introduces the ‘replication’ problem. Who manages the replication when the information changes in the source database server? How often? It’s more complex than it seems at first sight. And also introduces ‘another’ server to manage.



What can the tools for creating FHIR facades do for you?

We will explore tools for options 1 and 2.

Note 1: We will only discuss open source products. There are paid options in the market claiming that they allow you to do some of these and other chores without much programming, or in a more ‘assisted’ way.

Note 2: Not every library/tool supports ALL these features.

In general they allow you to concentrate on the **logic and implementation for your adapters,** taking care of the **‘FHIR plumbing’:**

- **Automatic generation of the conformance statement:** what resources does this server supports? The tools usually extracts this info from the adapters you implemented, and knows how to answer the /metadata request without additional programming.

- **Validation of resources:** some of the tools allow validation of the resources against the standard or specific profiles.

- **Routing of requests and parameters**: the tools take care of the HTTP request and response, and they will automatically route the request to the appropriate service, based on resource type and/or operation, including all the related parameters.

- **Logging**: the tools let you specify the logging mechanism and configuration.

- **Security**: the tools let you specify the kind of security you want to implement (none, basic, oauth2, Argonaut)

- **Format**: some of the tools take care of the request and response format (FHIR+XML, FHIR+JSON), some others are limited to JSON.

- **Error Handling:** most of the tools take care of the controlled response in case of errors (due to business logic, general validation or connection related problems). This includes responding with the adequate HTTP status.

- **Paging Management:** one of the more relevant features of FHIR is its search mechanism. And when there is great volume of data, clients may opt (or be forced to) paginate the searches. The complexity of paging is (sometimes) managed by these tools: calculating the quantity of total pages, providing links to the next and previous pages through named searches, etc.

- **Caching:** some of the tools allow the activation of caching mechanism to ease the main server if the requests are repeatedly hitting the same resources.

- **Support for different paradigms:** some of the tools only support REST and special operations ($). Other tools support messaging, transaction and batch processing.

- **Support for different FHIR versions:** most tools support multiple versions by configuration. Anyway, your specific adaptation may change accordingly.

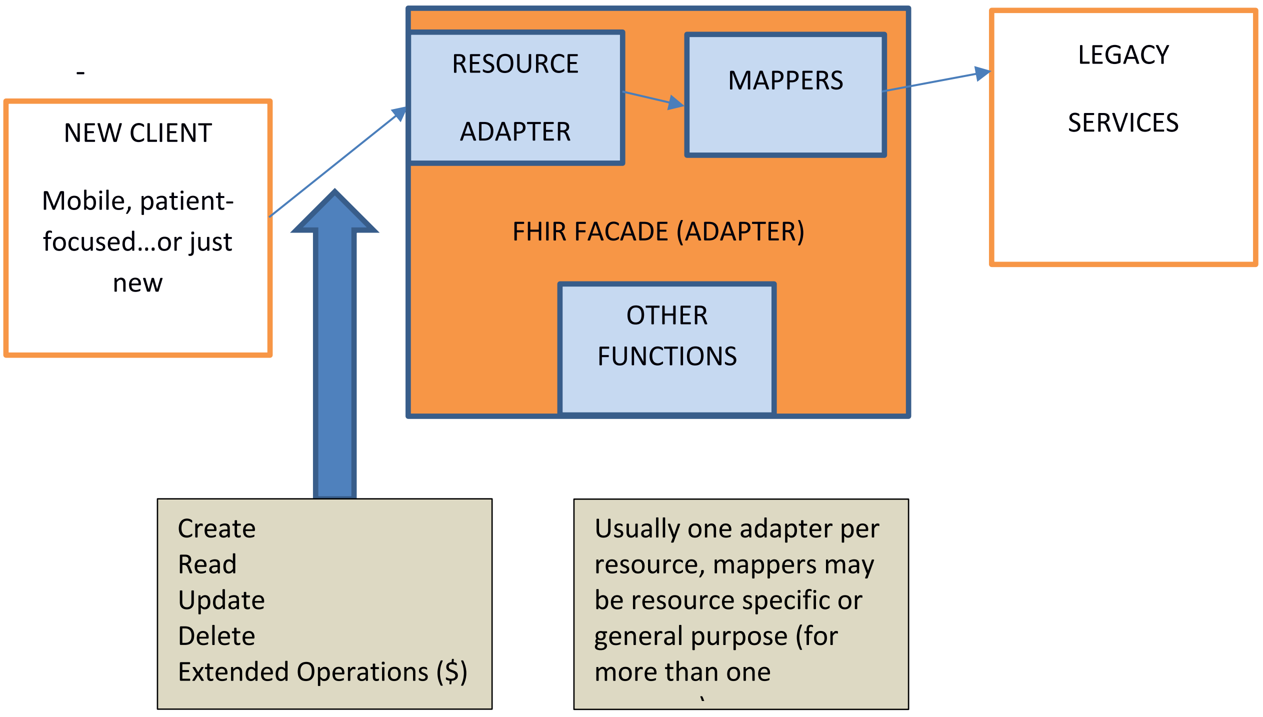
- **Advanced features:** some of the tools leverage the SMART-on-FHIR spec or GraphQL capabilities (this is out of scope for our course)

What is expected from you?

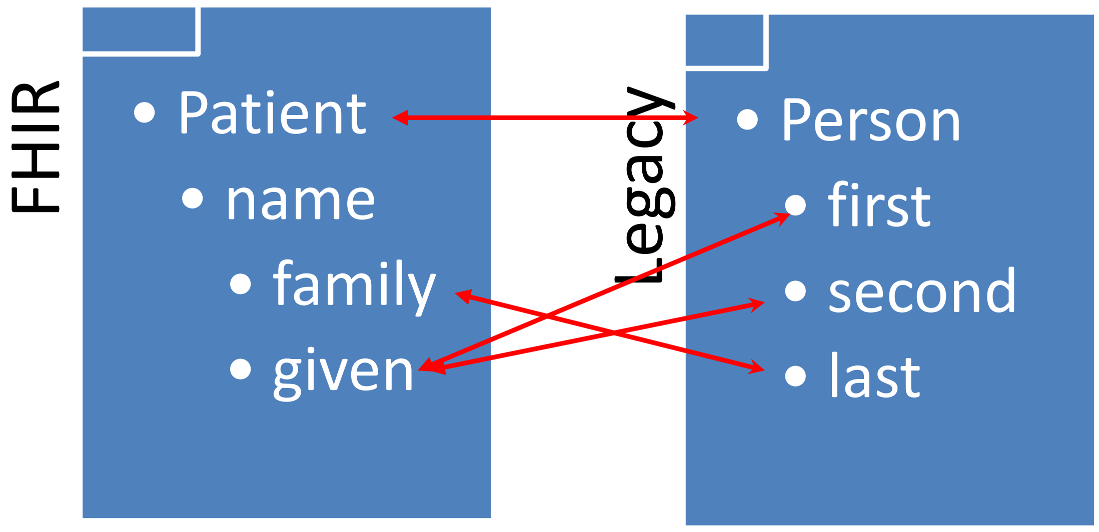
Usually, you will have to program “something” in the tool platform/language (if it’s a Java-based tool you will program in Java, if it’s a Node.js tool, you will program in node, etc.)

This “something” has more or less complexity depending on the distance from your model to the FHIR model, and the vocabulary dissonances: you will need to program adapters and mapper (transformations) for:

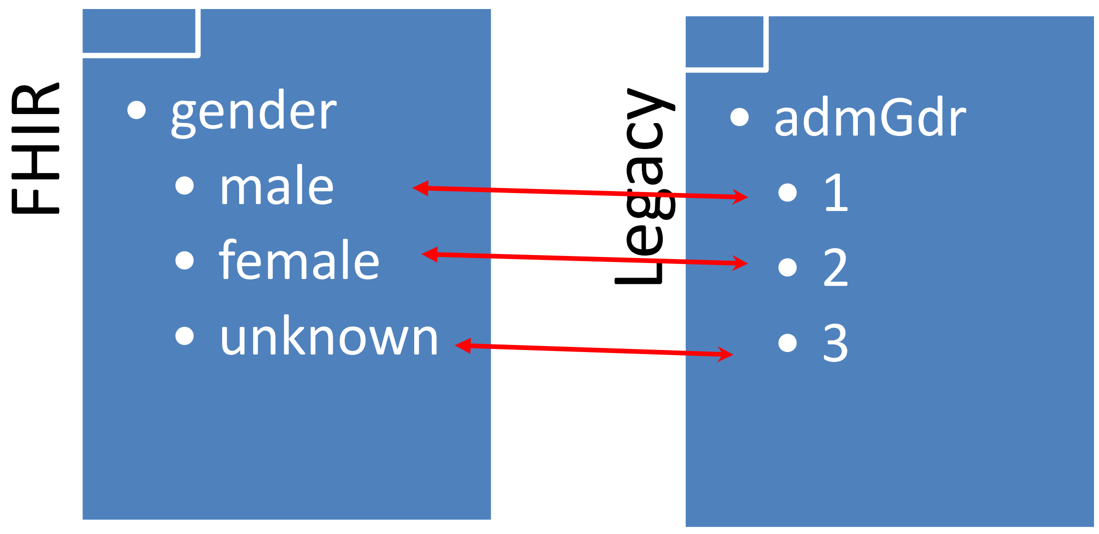
- **Route/Adapters:** this is how you implement each FHIR method or operation for your own project: how do I ‘read’, how do I ‘search’, how do I create a new resource? The tool will give you the resource and/or parameters, but you will be in charge of going to your legacy API or database and implement the method.
-


NEW CLIENT
Mobile, patient-focused…or just new
RESOURCE ADAPTER
MAPPERS
FHIR FACADE (ADAPTER)
OTHER FUNCTIONS
LEGACY SERVICES
Create
Read
Update
Delete
Extended Operations ($)
Usually one adapter per resource, mappers may be resource specific or general purpose (for more than one

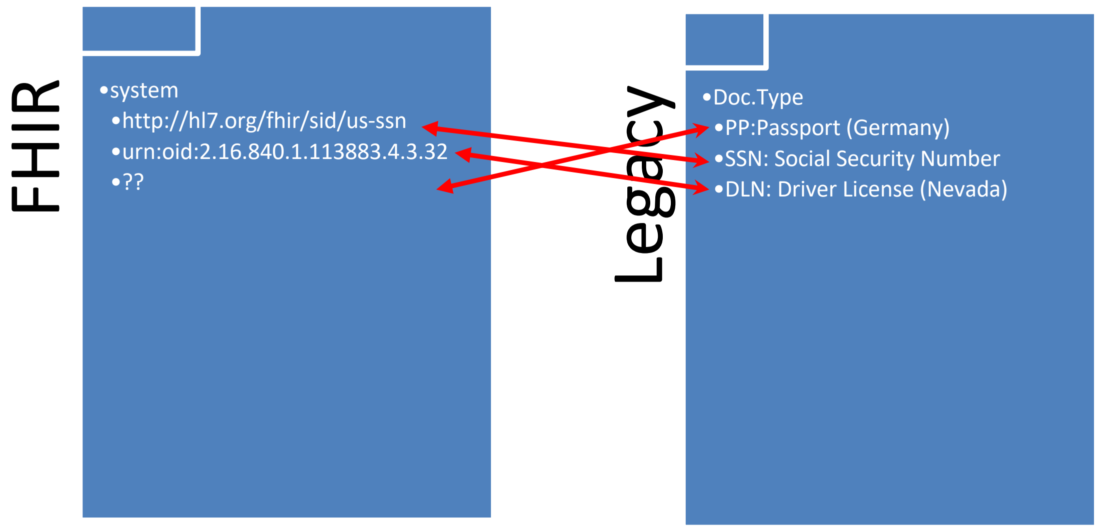
- **structures:** from your legacy data elements to FHIR resources and vice versa
  This picture shows a happy and simple scenario, but dissonances can be astonishing: more than one resource to one element, more than one legacy element to one resource, missing elements, repeating elements on one side but no repetitions in the other, etc.



- **code systems:** from your legacy code systems to your FHIR/IG requirements
  Again a happy and simple example. Or not.



- **identifiers:** from your legacy identifier to FHIR URIs or OIDs


FHIR
system
http://hl7.org/fhir/sid/us-ssn
urn:oid:2.16.840.1.113883.4.3.32
??
Legacy
Doc.Type
PP:Passport (Germany)
SSN: Social Security Number
DLN: Driver License (Nevada)

- **business rules:** enforce the validation of your business rules, and trigger appropriate responses in a FHIR manner.

You will also have to provide and configure if needed (<u>**always needed in production**</u>):

- **logging:** where is the log stored and which level of logging is required
- **security:** who can access the resources and how they prove their rights to access
- **caching:** where is the cache and how it is activated

The way that these artifacts (adapters, transformations, business rules, security, logging, etc.) are implemented depends on the platform:

- For Java: through the use of **annotation** and **interceptors**
- For node.js and .NET: through the use of **middleware** and **services**
  We will discuss more details in the codelabs for this unit.

Code Labs

The next sub-sections of this unit will expose how each FHIR Server tool allows you to implement some of the above mentioned functionalities:

Lang	Description	URL in this Course - Click to start now!
Java	HAPI FHIR Server	https://courses.hl7fundamentals.org/campus/codelabs.php?name=Codelab%20W3%20java&course=FHIR_INT_EN_2020_01#0
JS	Asymmetrik FHIR Server	https://courses.hl7fundamentals.org/campus/codelabs.php?name=Codelab%20W3%20JS&course=FHIR_INT_EN_2020_01#0
C#	Home Made C# FHIR Server	https://courses.hl7fundamentals.org/campus/codelabs.php?name=Codelab W3 CS&course=FHIR_INT_EN_2020_01 - 0

You don´t need to read **ALL** the Code Labs if you are not interested.

However, you **are required to fulfill the assignments for this unit**, developing or fixing a client in C#, Java or JS, so you need to know **at least** how to do what is required, by going through **one of them.**

And it is always good to learn something new.

Throughout the codelabs you will see three distinct icons:

The **‘example’** icon means that the following is a code example with actual values for the parameters it uses.
Some examples are not in-line with the text due to their length. You will need to refer to an external file.
Some of the long examples are also in github gist. When such an example is included, you will also see the github icon:

The **“recipe”** icon signals a “Recipe” or template. Recipes are code snippets with pseudo variables or objects.
After the code or pseudocode of the recipe there will be a table of the pseudo variables you will need to replace.
You need to replace these pseudo variables with your actual content.
Example <<ResourceName>> --> Patient

Sometimes to illustrate a method we use both examples and recipes, and sometimes only examples, depending on the subject complexity.

This week’s assignments

In this week will have formal “programming” assignments.

We will ask you to change or enhance an implementation of a Java, C#, or JS FHIR server.

You can try more than one language and/or track if you are able to, we will grade you with the maximum grade you obtained.

Unit Summary and Conclusion

FHIR is a ‘platform’ specification. In order to do real work, we need to use our desired platform to program our access to FHIR Servers.

We tried with this unit that you grasped common concepts that apply to all servers in FHIR and then apply them to the most used platforms / languages (Java,C# and JS)

Additional Reading Material

Information about FHIR

There are a number of places where you can get information about FHIR.

- The specification itself is available on-line at www.hl7.org/FHIR. It is fully hyperlinked & very easy to follow. It is highly recommended that you have access to the specification as you are reading this module, as there are many references to it - particularly for some of the details of the more complex aspects of FHIR.
- All the subjects detailed in this unit are deeply documented in the FHIR specification and implementation guides. If you can't explain something, this should be the first source-of-truth.
- The root HL7 wiki page for FHIR can be found at http://wiki.hl7.org/index.php?title=FHIR . The information here is more for those developing resources, but still very interesting. Some wiki information is more historical and may not reflect the most recent version of the specification.
- The team uses the FHIR chat (http://chat.fhir.org) as a place to answer implementation-related questions - and therefore have both question and answer available for reference.