FHIR Intermediate Course, Unit 2

FHIR CLIENTS
Reading Material

Course Overview

Module I: Implementation Guides

Most relevant FHIR Implementation Guides: Argonaut & IPS
Argonaut Development and Roadmap
Argonaut Data Query IG: Scope, Use Cases
Argonaut Provider Directory IG: Scope, Use Cases
IPS FHIR IG: Scope, Use Cases

Module II: FHIR Clients

General Guidelines for FHIR Clients
FHIR Clients in Java/JavaScript / C#

Module III: FHIR Facades

Why use FHIR Server Facade: your system on FHIR
Specific FHIR Servers (FHIR Facade)
Facade Use Case / Scenarios
Facade Architecture / Patterns
Where to put the FHIR Facade
System Integration / Integration Engine / Bus / Messaging
Facade in C# / Java / Node.JS [1 - Elective]

Module IV: FHIR Applications

Smart-On-FHIR
CDS-Hooks
Integration with Smart-On-FHIR / CDS-Hooks [1 - Elective]

Table of Contents

Unit Content and Learning Objectives

This unit is about developing FHIR Clients using reference libraries, and connecting to Argonaut/IPS compliant FHIR servers.

We will present the general concepts for all client libraries (how to perform common tasks in FHIR: connecting, accessing resources, etc.) and then describe how each library allows you to do each task.

So, this unit in fact contains 3 different sub-units on how to use each library (Java, .NET, JavaScript)

These sub-units have a set of micro-assignments. These assignments will not be graded and the solutions will be posted along the reading material. You can choose to try to solve them by yourself or just review and try the solutions)

After Unit 2, you will be able to:

- Describe the general structure of a FHIR Client: Connection, Security, Resource Model [Both tracks]
- Relate the content of the IGs to their related structures in selected FHIR computer languages implementations (JavaScript, Java, C#) using reference libraries [Both tracks]
- Produce a FHIR Client capable to connect to an Argonaut Data Query server, find a patient, receive a set of resources and display them to the user. [Track A]
- Produce a FHIR Client capable to connect to an Argonaut Provider Directory server, and find providers for a given specialty next to the patient location, and display the details to the user. [Track A]
- Produce a FHIR Client capable to connect to a FHIR/IPS server, find a Patient Summary, and display it to the user [Track B]
- Produce a FHIR Client capable to connect to a vocabulary server in order to retrieve a set of values for selections or value translations [Both tracks]

Overall Structure of a FHIR Client

In this section we will discuss concepts common to FHIR Clients in every language or platform.

> Remember that a 'client' in the FHIR world...may also be a server for another process, service, or component, and it's not necessarily 'user-faced' or 'presentation layer oriented'.
>
> **The only feature that defines an application or computer program as a FHIR client is that it will use a FHIR server to store (create, update, delete) or from which to read (search, read) FHIR resources.**
>
> So we will teach you just that: **how to use available libraries** to **read, search, create, update, delete resources**, and other relevant concepts for you as a FHIR Client developer (i.e. : **how to create transactions, documents, perform resource validation, parse and serialize resources, use terminology services**, and query a server for **conformance** information)
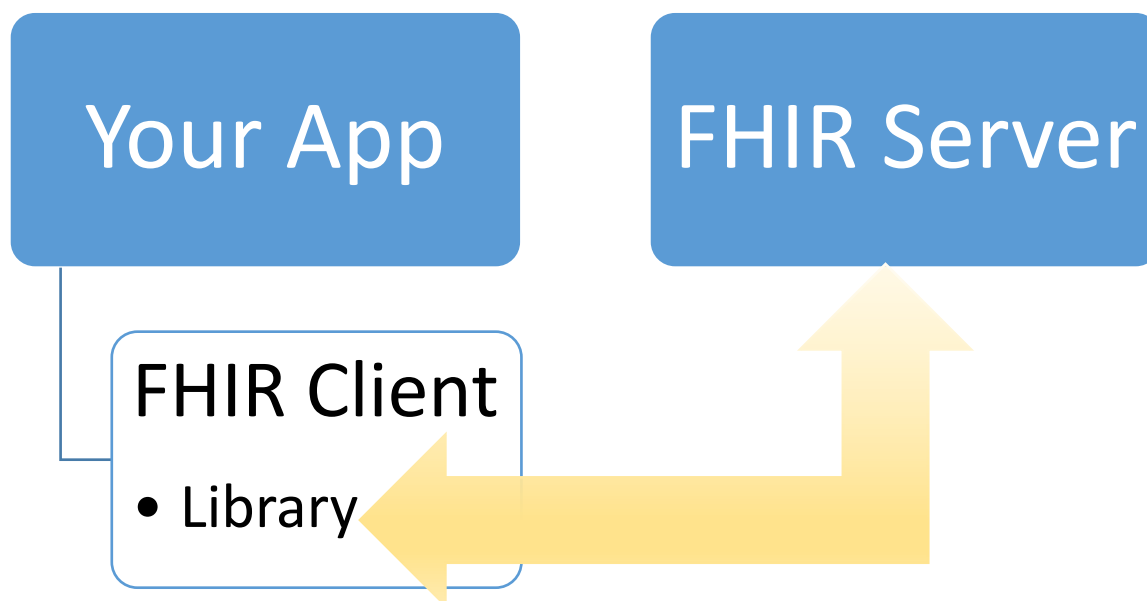
We selected one library for each platform (.NET, JAVA, JS) due to their popularity and/or scope.
You will find this document very brief, because most of the knowledge will be delivered through our **Code Labs**
Anyway, you can use the contents of this document to grasp a better understanding of what we are trying to achieve.
Code labs are interactive instructional tutorials, originally developed by Google that combines text and source code. See details at https://codelabs.developers.google.com/

We leveraged it by including small **"recipes"** so you can see what each concept involves in terms of coding, go through a real **example**, and try it yourself with something we call a **"micro assignment"**.

Note 1:
Our Micro-Assignments are not graded. They are just what we think you need to do in order to learn each concept. You will be graded by how you try to implement these concepts later, in the actual assignments for this unit

Note 2:
The solutions for the Micro Assignments are provided at the end of each code lab. So if you just want to peek at them and use them as a reference for future copying & pasting, it's up to you.

Note 3:
You don´t need to do ALL the Code Labs. If you are specialized in Java, and just want to learn how to do FHIR in JAVA, that's it. However, it is nice to learn a new thing once in a while.

Here, we will discuss each concept in general, and in our specialized Code Labs (C#, Java, JS) we will discuss how each selected library implements in each language/platform:

1. Connection

The first step for every FHIR client is to specify the server endpoint, headers and authentication. Some libraries allow specifying the FHIR version. Others are version specific.
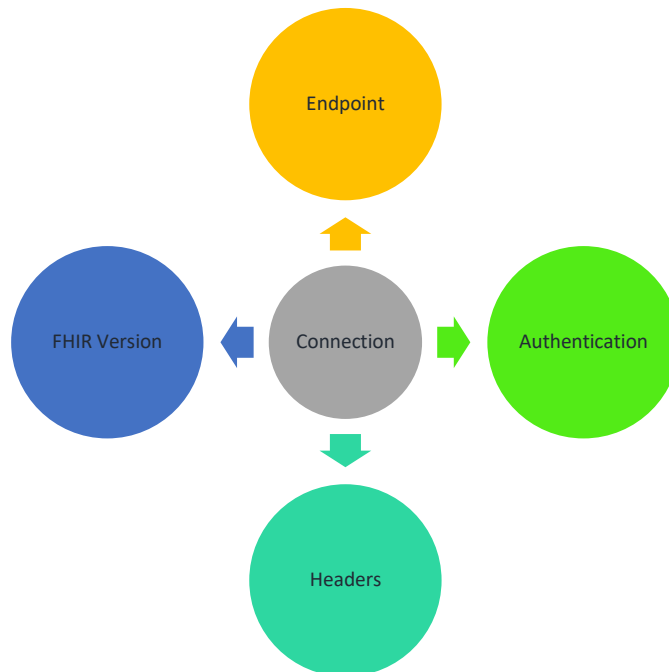
Endpoint

The endpoint is the address of the server without specifying any resource, it's also called the Server's FHIR address or URL, or also FHIR base endpoint. We mention all these terms because this is how the different vendors you may work with may use any of them.
All libraries have a way to specify the base endpoint for the server, usually when initializing the client.

Authentication

All client libraries allow to specify some kind of authentication strategy:
Most common parameters are:

oAuth2 (Smart-On-FHIR, Argonaut)

Bearer Token: token received from the authentication/authorization server. This token is included in all further calls.

Redirect: Client URL to call after successful authentication.

Scope: Scope (resources and operations -read/write-) the client wants to use.

Basic Authentication

Secret: secret string to be sent to the authentication server
Basic: for basic authentication. Involves sending a username and a password

Headers

All libraries allow adding additional headers to the request, like those defining request and response format (accept, content-type), additional parameters (x- headers), etc.

FHIR version or context

Some of the libraries are FHIR version-dependent, i.e. the library itself works only with a specific version of FHIR.

Some of them, let you define something called "context" for the client, and this context defines the version.

The later approach allows the same application to interact (at the same time) with servers using different FHIR version and even to implement FHIR version mapping.

2. Basic RESTful Methods
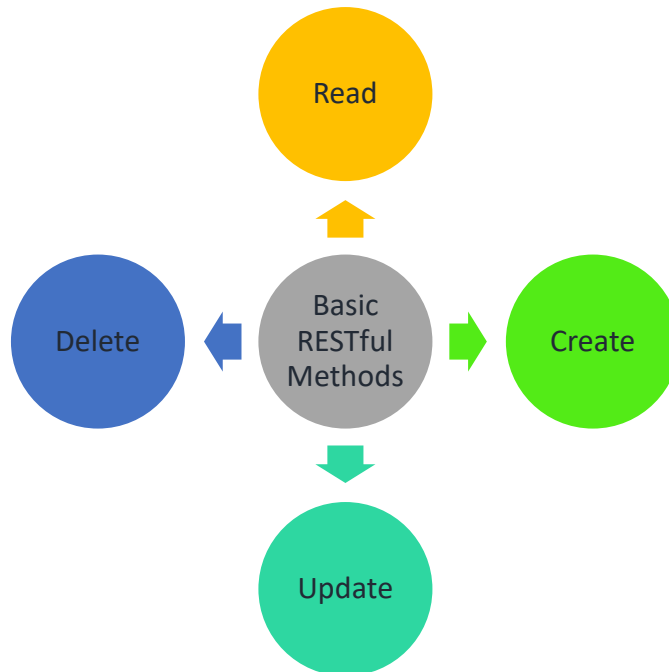
Read

All libraries allow direct reading of resources (direct **read**). Some of them allow reading specific versions of a resource (This is called **Version Read** or **vread**). Reading involves passing the resource type and the logical server id (this is different from 'searching', it's a direct GET. It's also important to be able to read a resource knowing the full URL (url read), to allow for distributed storage of resources.
If you do a direct read to the resource after deletion, the answer will be 410 (Deleted).
If you remember what you did with Postman in our previous courses, this would be
GET <endpoint>/resource/id

Create

Not all libraries allow the creation of resources.
Creating a resource involves
a. populating the resource elements – achieved by completing the resource model – see section below on 'Resource Model',
b. Invoking the create operation and
c. processing the response (successful creation/error).

If you remember what you did with Postman in our previous courses, this would be
POST <endpoint>/resource with the resource in the body of the request

See the advanced REST operations section below for Conditional creation of resources.

Update

Not all libraries allow updating resources (for instance, the original Smart-On-FHIR Argonaut scope was 'read-only')

Updating a resource involves

a) reading the resource current status -meaning in fact, the actual resource content-,

b) populating the changed resource elements – achieved by completing the resource model – see section below on 'Resource Model',

c) invoking the update operation and

d) processing the response (successful update/error).

See the advanced REST operations section below for Conditional update of resources.

If you remember what you did with Postman in our previous courses, this would be **PUT <endpoint>/resource/id with the resource in the body of the request**

Delete

Although rarely used (marking as no longer active is recommended), some libraries allow deletion of resources, because some use cases require this operation.

Server specific processing will determine if the deletion is processed or rejected, and how to handle the deletion internally.

If you do a direct read to the resource after deletion, the answer will be 410 (Deleted).

3. Advanced RESTful Methods
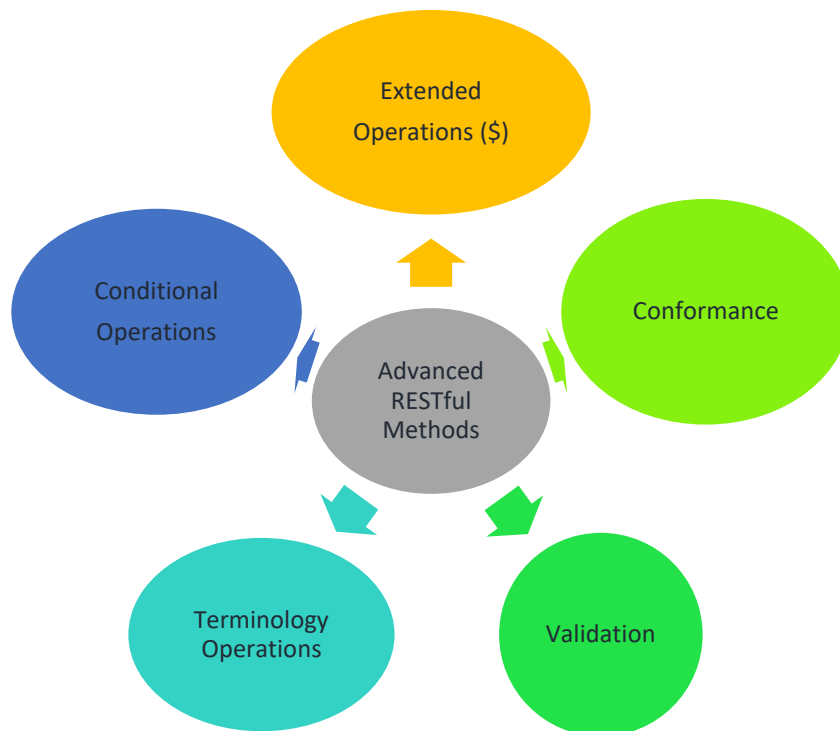
Conformance

This should be the first call that your client makes to any server. It will allow your client to find out if this specific server is able to process the operations you need and supports the profiles you are using.

All libraries allow requesting the conformance resources.

In fact, some libraries read and process the metadata resource as soon as the client is connected (you can test this by tracing the library communications with the server)
And some others facilitate traversing the conformance resource to allow answering some basic questions about the server: does this server support searching by city? Does this server implement versioning on the Patient resource, etc.

If you remember what you did with Postman in our previous courses, this would be:
GET <endpoint>/metadata

Extended Operations ($)

These are operations 'beyond REST'.

They may be defined for specific resources, instance specific - a specific instance of a given resource class) or for the whole server.
Terminology specific operations will be discussed below, under 'Terminology operations'.

One example of a server wide operation would be:
_history (applied to all the resources)
This operation requires the **since** parameter.

One example of a instance specific operation would be:
$everything (applied to a specific patient)
If you remember what you did with Postman in our previous courses, this would be
GET <endpoint>/Patient/<id>/$everything

Some of these operations are not directly supported by the libraries we've chosen. In these cases, we teach how to implement them using standard libraries (because, after all, FHIR Servers can be used with no libraries, just plain HTTP/S calls).

Conditional Operations

This kind of operations allows you to only create or update an specific resource if it does or does not exist previously, based on a search criteria (usually, an identifier).

Most libraries allows this. Usually, this is used as part of a transaction involving several kind of resources to avoid multiple trips to/from the server.

Terminology Operations

Terminology operations are a set of specific $operations beyond RESTful get/search, intended to implement **terminology services** for concepts defined in given vocabularies.

Target vocabularies need to be defined and stored (or façaded) as ValueSet, CodeSystem resources.

We will treat them separately because they are important in any project.

We will only cover the most common usual operations in this course, enough to populate lists or validate codes.:

Operation Name	Resource	Parameters	Description
$validate-code	ValueSet	url, system, code	Validate a code against a ValueSet. url=canonical identification of the ValueSet. system,code: code to validate
$expand	ValueSet	url, filter	Get a (filtered) list of concepts from a ValueSet ("an expansion"). Useful for populating lists or implementing smart searches. Use url to specify the specific subset.
$validate-code	CodeSystem	url, system,code	Validate if a code exists in a given CodeSystem
$lookup	CodeSystem	system, code	Get details on a specific code (properties, denominations, hierarchy) - depends on the code system.

Keep in mind:

Not all FHIR servers support terminology operations.

Not all FHIR servers supporting terminology operations have loaded or support SNOMED or its expressions, LOINC and/or the HL7 vocabularies.

Operation $expand for ValueSet
Expand sometimes requires transforming a definition of the valueset into an actual list of concepts, through expression processing. Not all servers support this kind of expansion. Some servers need the ValueSet defined by extension (every concept is defined in the ValueSet)
This server supports ValueSet/$expand for SNOMED CT valuesets intensionally defined:
https://snowstorm-fhir.snomedtools.org/fhir
This server supports ValueSet/$expand for LOINC valuesets:

https://fhir.loinc.org

Operation $validate_code for ValueSet
This operation allows to validate a given code (with its codeSystem) against a specific ValueSet

Note: SnowStorm currently does not support this operation so you need to find another server.
This server supports ValueSet/$validate_code for SNOMED CT valuesets:
https://valentiatech.snochillies.com
This server supports ValueSet/$validate_code for LOINC Valuesets, but works (as of Nov 5, 19) in DSTU3:
https://fhir.loinc.org

Some interesting terminology services where you can practice with our micro assignments:

UMLS

Includes access to several code systems including SNOMED CT, LOINC, C-CDA vocabularies, HL7, etc.
https://cts.nlm.nih.gov/fhir/
Requires a free account and basic authentication using your user name and password

LOINC

LOINC publishes its own FHIR enabled server for LOINC terminology.
https://fhir.loinc.org/
Requires a free account and basic authentication using your user name and password
A good primer on how to use LOINC with FHIR is here
https://danielvreeman.com/loinc-on-fhir/

SNOMED CT:

Here is a curated list of SNOMED CT FHIR Enabled servers
https://confluence.ihtsdotools.org/display/FHIR/Features+of+Known+Servers
Snowstorm is a FHIR enabled Snomed CT server developed by SNOMED International. (You can even download Snowstorm to setup your own terminology server)
https://snowstorm-fhir.snomedtools.org/fhir
A good overview on how to use Snowstorm with the FHIR API is here
https://github.com/IHTSDO/snowstorm/blob/master/docs/using-the-fhir-api.md

Resource Validation

As recommended in Postel Law ("Be liberal when receiving and conservative when sending"), it's recommended to validate your resources against the FHIR specification and your specific template or profile before sending it.
Most libraries enable some kind of resource validation either against a server or locally, but not all servers support resource validation, and not all libraries support all variants of local resource validation

Options as a FHIR client, with a resource you have created:
1) Send your resource to a server, and let the server validate it.
2) Send your resource to a server, and let the server validate it (against a profile)
3) Validate a resource locally
4) Validate a resource locally against a profile

Options as a FHIR client, with a resource you receive
5) Validate the received resource locally
6) Validate the received resource locally against a profile

4. Resource Model

Some libraries facilitate access to the content of the resources through something we (and the library creators) call 'the resource model' : a set of classes and methods to set or get the values of the resource elements, and also to instantiate them.

These classes and methods provided by the libraries also make your life as a developer easier because they enable **Intellisense -**or intelligent code completion (the ability of your IDE or editor to enable context based code completion and early validation -i.e. before compile time, way before run time).

This happens in Java and .NET because they are strongly typed. In JS, unless you incorporate some experimental TypeScript FHIR models, you are on your own, accessing directly to the objects through the object name.

Structures

If you want to have an structural view of what we talk about when we mention "the FHIR resource model", it can be depicted by the graphic below (this taxonomy is ours and was created to understand what the FHIR libraries need to do for you).

Libraries for FHIR Clients (and for Servers too but we will talk about this later!) need to assist you in gaining read/write access to all the boxes in this diagram.
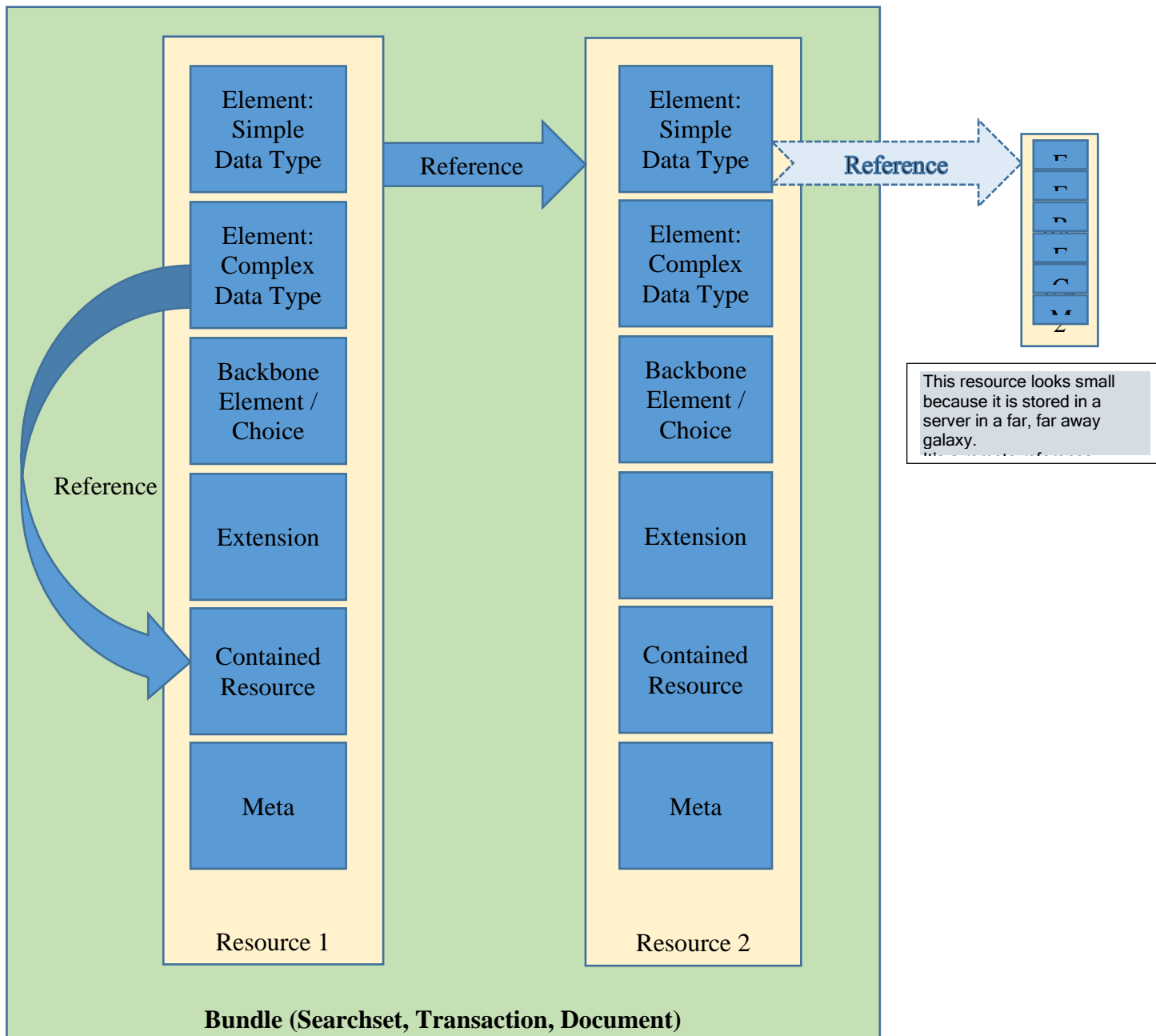
- **Elements in Resources:** can be defined by the FHIR spec as

  o **simple data types** (like strings, booleans or numbers),

  o **complex data types** (addresses, names, identifiers, attachments, codes),
  o something mysterious called '**Backbone Element**' which is just an element holding several complex or simple elements.

  o **'Choices',** they can include elements from a menu or option list of data types, so they are easier to create (just include an element with the correct name and datatype) but difficult to process (you need to know first the specific data type of the element in the specific instance of the resource you received, and then process it!)

- **Extensions:** you need to create / add to your resources -or to any element in your resources - both simple and complex extensions, and process them if you receive a resource holding extensions and you are interested in their content.

- **Contained Resources:** some resources contain other resources. You need to create the new resource and include it in the container, and also if you receive a contained resource, follow the internal reference and process it.

- **Meta:** this part of the resource it gives you version (date, identity) , profile (maybe more than one), and security information

- **References:** you need help in setting up and/or process three types of references between resources: **contained** (references to contained resources), **external** (reference to other resources in the same or other servers), and **intra-bundle** (references between resources in the same bundle)

- **Bundles:** you need to create and/or process bundles. Mostly, as a client, you will generate "transaction" or "document" bundles, and process "document" or "searchset" bundles.

Fluent vs Structured

When creating elements, backbone elements or resources, we will discuss for each library the fluent vs structured aspect, we experienced three levels of fluentness:

Structured: you create intermediate objects or elements, and then assemble them into the resource or element by 'adding' them.

Example (in an invented language)

```
…
ElementType MyElement=new ElementType(att1:something,att2:"another thing");
MyResource.setElement(MyElement);
AnotherElementType AnotherElement=new AnotherElementType(att1:someInt,att2:"other another thing");
MyResource.setAnotherElement(AnotherElement);
…
```

Fluent: you can chain the function calls as you need them to create and populate your resources.

Example (in an invented language)

```
…
MyResource.setElement(att1:something,att2:"another thing").setAnotherElement(att1:someInt,att2:"other another thing");
…
```

Direct: you build the resource like you would in Postman, accessing the objects directly (or almost). If you look at the resulting code, the structure is pretty similar to the object (even to the JSON object).

```
…
MyResource=
{
Element:
  {att1:something,att2:"another thing"},
AnotherElement:
  {att1:someInt,att2:"other another thing"}
}
...
```

We do not favor any of the strategies, and the libraries usually try to let you choose - if it's possible for the language - but…you can evaluate how many **boilerplate code** you write for each option (on one side) and how **readable, maintainable and protected from errors** is the code (on the other side).
Note: these are our own definitions of fluentness, and are not a "documented" feature, specially the last level, just something we observed using all the libraries.

Repeating Elements / Cardinality

There is another level of complexity: other nuance we need to take into account, is **cardinality**: how to include elements that CAN be repeated.

If an element **can be repeated according to the FHIR spec** (example: several identifiers, several name parts, several names, several codes for coded elements), i.e. anything [0..n] or [1..n] in the **Card.** column below, then it will be represented as an array in the object representing the resource, component or element. For these cases, some libraries use a specific method, something like `object.array.add(new_item)` or `object.array.push(new_item)` –

Everything can be an array (of course depending on the actual resource and data type definition in the FHIR spec)

Structure

Name	Flags	Card.	Type	Description & Constraints	?
Condition	I TU		DomainResource	Detailed information about conditions, problems or diagnoses + Guideline: Condition.clinicalStatus SHALL be present if verificationStatus is not entered-in-error and category is problem-list-item + Rule: If condition is abated, then clinicalStatus must be either inactive, resolved, or remission + Rule: Condition.clinicalStatus SHALL NOT be present if verification Status is entered-in-error Elements defined in Ancestors: id, meta, implicitRules, language, text, contained, extension, modifierExtension	
identifier	Σ	0..*	Identifier	External Ids for this condition	
clinicalStatus	?! Σ I	0..1	CodeableConcept	active | recurrence | relapse | inactive | remission | resolved ...ed)	
verificationStatus	?! Σ I	0..1	C...	... | confirmed | refuted | entered-in-error	
category		0..*	CodeableConcept	problem-list-item | encounter-diagnosis Condition Category Codes (Extensible)	
severity		0..1	CodeableConcept	Subjective severity of condition Condition/Diagnosis Severity (Preferred)	
code	Σ	0..1	CodeableConcept	Identification of the condition, problem or diagnosis Condition/Problem/Diagnosis Codes (Example)	
bodySite	Σ	0..*	CodeableConcept	Anatomical location, if relevant SNOMED CT Body Structures (Example)	
subject	Σ	1..1	Reference(Patient | Group)	Who has the condition?	
encounter	Σ	0..1	Reference(Encounter)	Encounter created as part of	
onset[x]	Σ	0..1		Estimated or actual date, date-time, or age	
onsetDateTime			dateTime		
onsetAge			Age		
onsetPeriod			Period		
onsetRange			Range		
onsetString			string		
abatement[x]	I	0..1		When in resolution/remission	
abatementDateTime			dateTime		
abatementAge			Age		
abatementPeriod			Period		
abatementRange			Range		
abatementString			string		

Cardinality

Data Types

The most commonly used data types beyond simple ('primitive') ones in FHIR are: identifiers, person names, addresses, dates, attachments and coded values (simple codes or concept representations)

We will include a brief reminder of the usual structure and a JSON example, but you need to remember them from our previous FHIR courses (Don't you? ☺).

For the whole data type spec (all the data types ,all their components), please refer to:
https://www.hl7.org/fhir/datatypes.htm

> Please note in our brief structure and example, that the Data Type begins with a capital letter 'Identifier' and the element in the instance, begins with a regular letter ('identifier'), because "The element named **identifier** in the xxx resource is of the **Identifier** data type"

Identifiers

Identifiers usually have two parts: **system** (which application assigned this specific identifier) and **value** (the actual identifier generated for this instance). Some implementation guides also require **use** and/or **type** (what kind of identifier is it? Medical Record Number? National Patient Id?)

Identifiers apply to any real or virtual entity (patients, locations, specimens, physician, organizations, etc.) and usually to acts too (orders, observations, etc.)
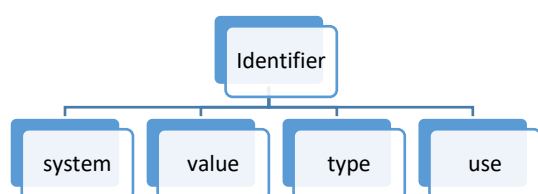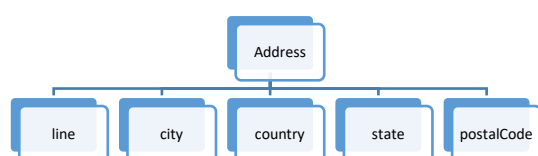


```
"identifier": [
    {
       "use": "official",
       "system": "http://citizens-id.gov/citizens",
       "value": "69999999-I"
    }]
```

Addresses

Addresses in FHIR have potentially lots of parts, but we will include only those required in our IGs for the sake of brevity.
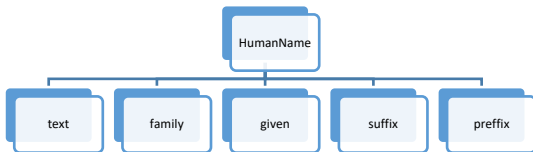


```
"address": [
    {
       "line": [
         "99999 Patient Street"
       ],
       "city": "Ann Arbor",
       "state": "MI",
       "postalCode": "48103",
       "country": "USA"
    }
    -
```

Human Name

Person names have different parts but in Core FHIR R4 the only parts included are Family Name (which does not repeat) and Given Name (which can repeat), so if you need to include more than one Family Name you will need to resort to an extension.
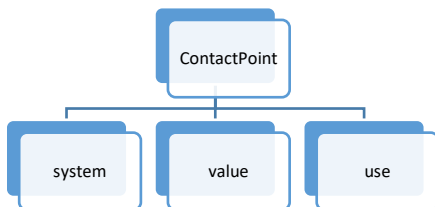


```
"name": [
    {
      "use": "official",
      "family": "Everywoman",
      "given": [
        "Eve"
      ],
      "prefix": [
        "Ms."
      ],
      "suffix": [
        "III"
      ]
    }
  ]
```

Email and Telephone Numbers

Most IGs require at least telephone and email addresses for patients and providers.
This is implemented using the ContactPoint data type, which allows to specify the system (phone, email) and the actual address (number, url, etc.), and also the use and validity period.


```
"telecom": [
    {
      "system": "phone",
      "value": "(777) 555-9999"
    },
    {
      "system": "email",
      "value": "eve@everywoman.com"
    }
  ],
```

Dates and Times

There are three different FHIR data types for dates and times: Instant, Date, and DateTime. Not very complex, but we include an example to show you what is expected from you.

```
"birthDate": "1968-07-23"


"deceasedDateTime": "2019-02-13T10:30:00-03:00"
```

Quantities

For quantities you need to express both the actual quantity and the unit of measure – unit should be from the http://unitsofmeasure.org system.
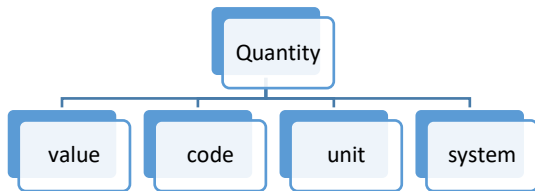

```
"valueQuantity": {
    "value": 6.3,
    "unit": "mmol/L",
    "system": "http://unitsofmeasure.org/",
    "code": "mmol/L"
  }
```

Coded Values

Simple Coded Items (code)

For **simple coded items where the binding strength is 'required'** (those defined by the FHIR spec), some libraries create the corresponding named "enum" (Enumerated Data Type, enabling validation and Intellisense, good for decreasing your bug count!)

```
"gender": "female"
```

Codeable Concepts (coding)

For codeable concepts you will need to specify both the code and the system (CodeSystem) where the concept is defined. You can also include the display element (the description of the code). Watch out: coding may repeat!
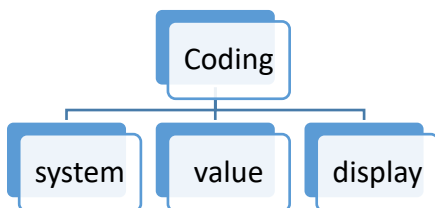

```
"maritalStatus": {
    "coding": [
      {
      "system":
"http://terminology.hl7.org/CodeSystem/v3-
MaritalStatus",
        "code": "W"
      }
    ]
  ,
```
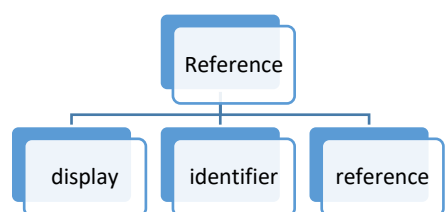
References

More details about FHIR resource references: https://www.hl7.org/fhir/references.html

You can make a reference to:
- to a resource in another server (using reference: full URL of the referenced resource),
- a resource in the same server (just the Class Name and Id, like reference: "Patient/20192")
- a resource in the same Bundle, referencing the full URL of the entry holding the resource (reference: "Patient/{GUID}"), or
- finally, to a Contained resource by including the XML REF of the contained resource id (Reference: "#MyResourceId")

If you don´t know the actual reference but you only know the business identifier for the resource, you can just include the identifier element.

You can always add the display for a human readable reference.

```
                                      "managingOrganization": {
         ┌──────────┐                     "identifier": {
         │ Reference │                         "system": "http://npi.org/id",
         └──────────┘                         "value": "999999"},
    ┌────────┬────────┬────────┐              "display": "Ann Arbor Gen.Hospital"
  display  identifier reference          }
                                      }
```

```
"subject": {                          "subject": {
    "reference": "Patient/12371",     "reference":"http:/myserver.com/fhir/Patient/12371"
    "display": "Eve Everywoman"        ,"display": "Eve Everywoman"
  }                                    }
```

```
"entry": [                            "device": {
        {"reference": "Observa-       "reference": "#MyDevice"},
tion/aa11a2be-3e36-4be7-b58a-         "display": "LabAnalyzer 3000"}
6fc3dace2741"}                        }
]
```
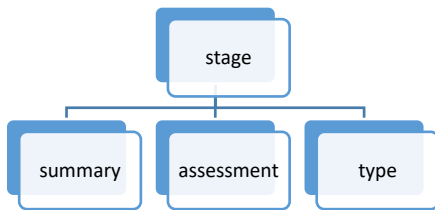
Narrative

Narrative is a significative part of the resource content, and should be always populated. The bad news is that almost no library implement automatic, template based completion of the XHTML Narrative for each resource, so you are mostly on your own there. Only FHIR HAPI for Java has some automated methods to template the generation of the text based on the content of each resource.

Backbone Elements

Beyond simple and complex data types there is this construct called 'Backbone Element'. Some libraries allow you to instantiate them, and get or set their value. It's like a more complex data type, anyway.


```
"stage":[{
"summary":
{"coding":
[{"system":
"http://snomed.info/sct","code":"385356007"}]},
"assessment":
[{"reference":"DiagnosticReport/201993"}],

"type":{"coding":[{"system":"http://snomed.info/sct"
,"code":"422375001"}]}
        }]
```
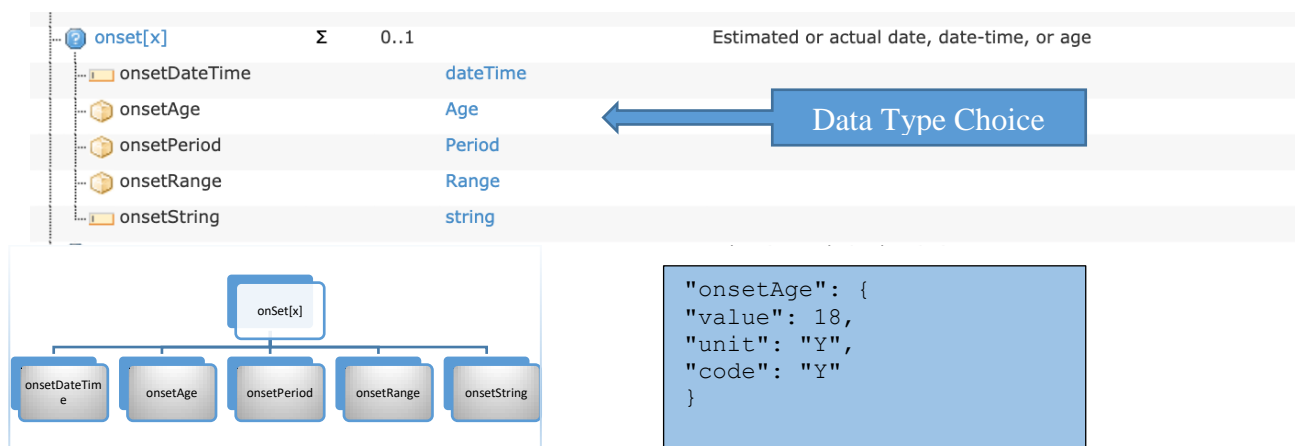
Choices

Choices are special elements which can be valued when generating the instance with a specific data type from a list defined by the spec. In a computer program, this may involve interrogating the object to see which data type it holds, or even asking for a specific data type we are interested in, and discarding if it is not present.

In fact, the FHIR spec presents two types of choices:

a) **Reference Choice**: choices between references, like the one we can see in the previous page in **assessment**. The data type is reference, but the reference itself is not to only one resource class, but to a list of options.

b) **Data Type Choice**: a kind of 'generic' data type, denoted by ElementName[x], x can be instanced only one as of one of the choices.

Depending on how the language manages these nuances, the library may provide you with accessor and creator methods for choices.

In strongly typed languages like C# and Java, the challenge to process choices derives from the fact that you may be unaware of the specific type of the element until you process the actual instance of the resource, so you will need to be sure that you cover all the expected cases, or believe in the restrictions of the Implementation Guide ("example: we will only generate and process onsetDateTime for onset")

Extensions

More details about FHIR extensions: https://www.hl7.org/fhir/extensibility.html

Extensions are fundamental to ensure adaptability of the FHIR spec for your use case.

Any element in FHIR can be extended, at the element, resource or data type level.

Your extensions should be documented as part of your implementation guide (the extension defini-tion includes the intent – where it belongs and what do you want to include there? - data type of the extension, and the vocabulary for coded elements).

Extensions can be simple (just an URL and a data item) or complex (containing other extensions)

Usual chores regarding extensions are creating them, adding them to any resource or element, test-ing for any/some extension existence in an element, and extracting the values

Some libraries implement special features to facilitate these chores.
Since extensions always can repeat (there is no limit to the quantity of extensions included in an instance or element), usually they require something like `ele-ment.extension.add(extension)` or `element.extension.push(extension)`
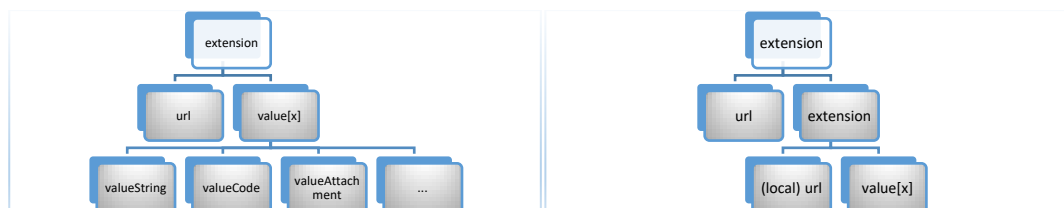

```
extension: [{
        "url": "http://hl7.org/fhir/us/core-
r4/StructureDefinition/us-core-ethnicity",
        "extension": [
          {
            "url": "ombCategory",
            "valueCoding": {
              "system": "urn:oid:2.16.840.1.113883.6.238",
              "code": "2186-5",
              "display": "Not Hispanic or Latino"
            }
          },
          {
            "url": "text",
            "valueString": "Not Hispanic or Latino"
          }
        ]
      }]
```

Transactions

More details about FHIR transactions: https://www.hl7.org/fhir/http.html#transaction

FHIR Transactions are a way to reduce round trips to the FHIR server when storing several related resources. They also ensure that all the resources are stored in the server successfully or none does (there is also a 'batch' bundle defined in FHIR where the processing of each resource is treated separately, but we will not discuss it in this course).

We will only explore how to submit a transaction to a FHIR server using any of the libraries.

The mechanism to submit the transaction is as follows:
- **Create** the transaction bundle
- **Add all the resources** to entries in the bundle
- Entries hold the resource and the operation or method to perform (-conditional- create or update).
- Relate the resources between them using temporary identifiers (valid and unique inside of the transactions)
- **Submit (POST)** the transaction bundle to the server FHIR endpoint
- **Process the results** (what happened to each entry)

Some libraries implement special methods and data types to facilitate some of these chores (example: submit the transaction bundle to the server and process the result).
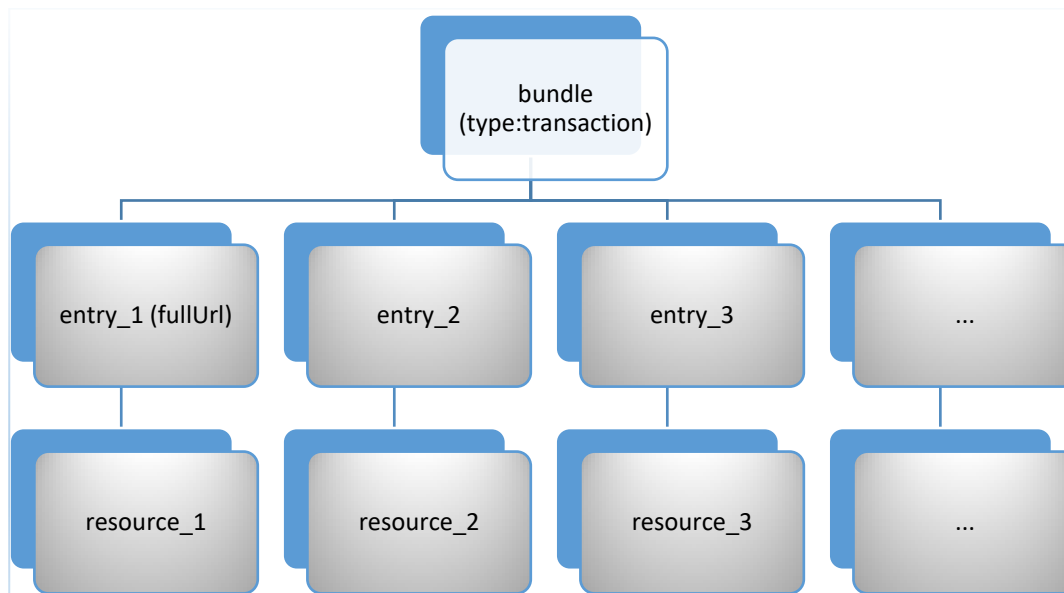
Documents

More details about FHIR documents: https://www.hl7.org/fhir/documents.html

FHIR Documents are special bundles with type="document", and a first resource "Composition"
The Composition carries the metadata about the document (creation date, type, reference to the diverse participants: author, patient, etc.), and each one of the narrative sections of the document. These sections themselves have also a link (reference) to the actual entries carrying the coded clinical information (observations, procedures, medication requests or statements, etc.)

We will only explore how to (as a client) create a FHIR document (i.e. an IPS document), and how to process it using the selected libraries.

The mechanism to create the document bundle is bottom-up (you need to create the "smaller" items in order to create the "bigger" ones) is as follows:
- Create all the entries and referenced resources (Patient, Organization, etc.).
- Create the sections with the text and the reference to the entries
- Create the Composition with the sections
- **Create the Bundle with the Composition and all the other resources as entries**
- **Submit (POST) the bundle to the server FHIR endpoint for bundles**
- **Process the results** (what happened to my bundle?)
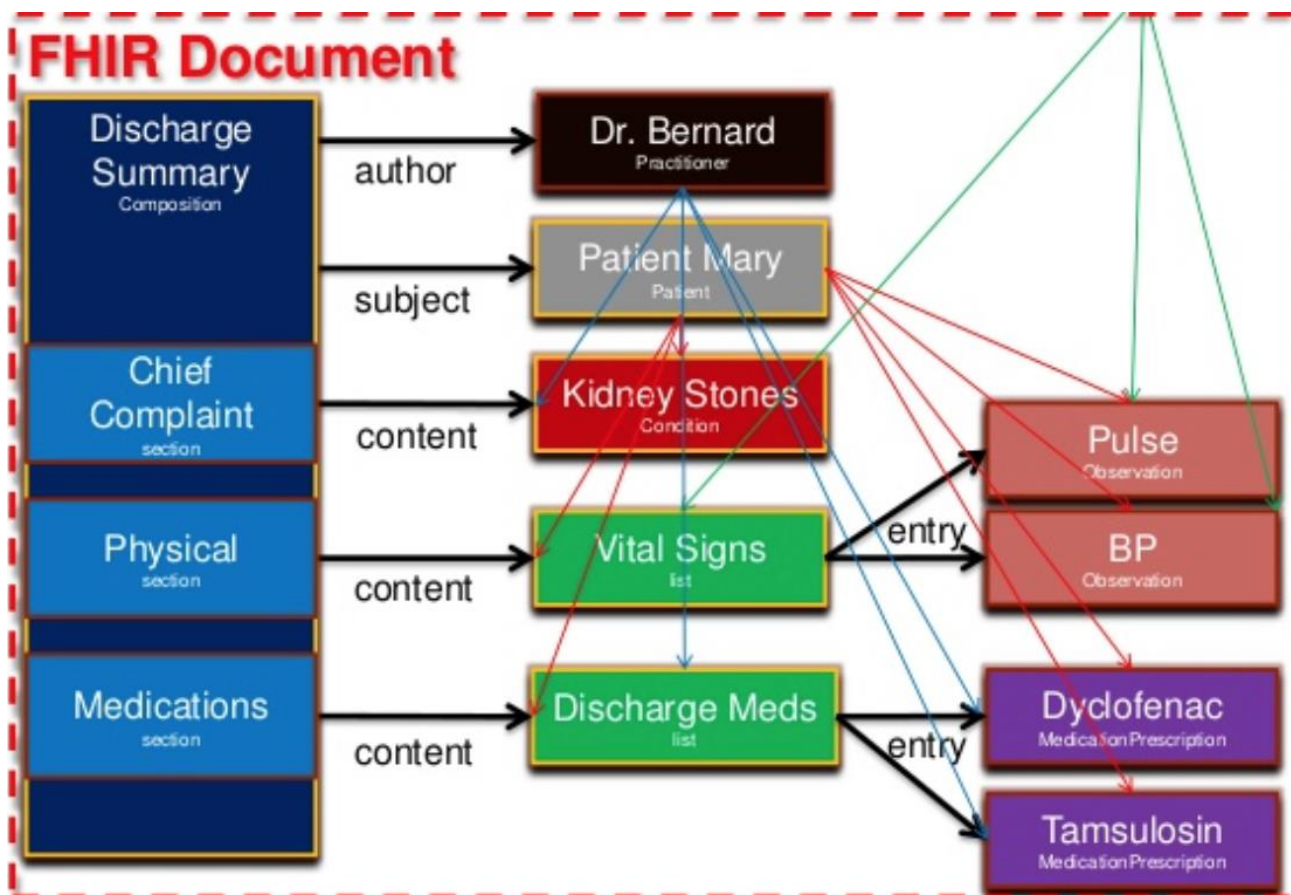
Contained Resources

When you have information that you need to reference from other resources, but you don't have enough information to be able to create a resource for it, you can use a contained resource. Perhaps you want to store just the name of a hospital, which is useful, but not enough to uniquely identify an organization (there may be two in the world -or even in your region- with that name). Contained resources are a way to allow small fragments of information to still be used as referenceable data. They also provide a way to create resource types (e.g. Organization), but not need the system to save and retrieve them as independent data, with a specific URL endpoint - which may be overkill for some simple data.

We will explore how to include contained resources and how to process them, using the selected libraries.

5. Common tasks
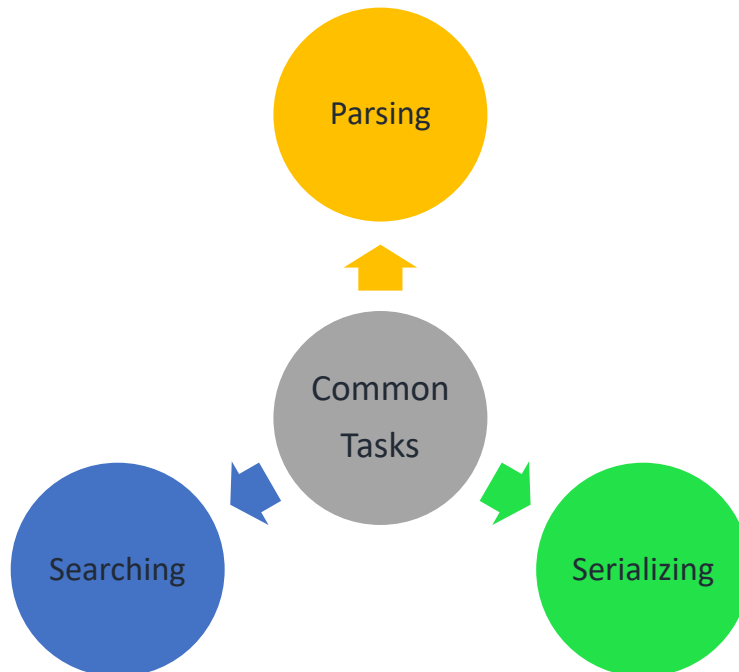
Parsing

A typical journey for a FHIR app includes receiving information in a stream format (file or download), maybe as XML as result of a transformation from the primary source, and converting it to a FHIR object for further processing (traversing, indexing, etc.).

This process, along with the validation of the received data format and syntax, is called "Parsing". We also need to parse the results that we receive as HTTP into an object in order to understand and process what happened with our requests.

We will study how to parse a string into a FHIR object using the provided libraries for each platform.

Serializing

As important as parsing is the other way around…what happens if I want to store a FHIR resource as a string or byte stream? What do I need to do to save it as XML in order to send it to an engine that only understands XML?

This process is called "Serializing". Although this is trivial for JSON (at least in JavaScript if it's already in FHIR format), it's not the same for every platform and every format.

We will study how to serialize a FHIR object into a string using the provided libraries for each platform.

Searching

Finally, we will explore searching in FHIR, how each library allows you to:

- Specify the search parameters
- Process the search results
- Implement paging and caching

Search Parameters

The most commons parameters for searching are the defined by each resource in the FHIR resource spec. Remember that each implementation guide and even each server can define which parameters it will support.

Compartment Search

Other relevant parameter for resource searching are compartments. Compartments are special sets of resources defined by the FHIR spec. They allow retrieval and access control of a group of resources instead of a specific class. Currently, the compartments are Patient, Encounter, RelatedPerson, Practitioner and Device.
The syntax for compartment search is GET baseurl:/[CompartmentClass]/[id]/[ResourceClass]
This will bring all the resources of the ResourceClass related to the CompartmentClass.
We had not use this FHIR search, but we include it here because it may be required for some regulations.
See more details here: https://www.hl7.org/fhir/compartmentdefinition.html

Included Resources in Search

Other interesting option for reducing the network delay produced by several queries and trips back and forth to the server is to include other resources related to the one we are searching.
This parameter is similar to the JOIN command in SQL and instruct the server to also return specific resources in the same bundle
Examples from the spec:
The medication request and the referenced patient
The medication request and all provenance resources referencing the specific Medication Request

```
GET [base]/MedicationRequest?_include=MedicationRequest:patient

GET [base]/MedicationRequest?_revinclude=Provenance:target
```

More details here: https://www.hl7.org/fhir/search.html

6. Code Labs

The next sub-sections of this unit will expose how each FHIR library allows you to implement all these functionalities:

Lang	Description	URL in this Course
Java	Our HAPI FHIR Client Handbook	Codelab W2 java
.NET	Our .NET FHIR API Client Handbook	Codelab W2 NET
JS	Our JS FHIR API Client Handbook	Codelab W2 JS

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 Java/.NET 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, objects** or **data type names.**
After the code or pseudocode of the recipe there will be a **table of the pseudo variables you will need to replace.**
Pseudo variables are signaled like this: **|PseudoVariableName|**
You need to replace these pseudo variables with your actual variable, class or attribute names.
Example |ResourceName| --> Patient

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

This **"Micro Assignment"** icon signals a "Micro Assignment". These Micro Assignments are meant for you to try the recipes and examples presented through the section.
Remember that Micro Assignments:
a) **Are NOT graded**, and we will provide the solutions along with this reading material.
b) Are **NOT the assignments for this unit**. These Micro Assignments are just for you to learn small concepts, one at a time.

This week's assignments

In this week will have formal "programming" assignments .We will ask you to change or enhance some programs using the Java, C# or JS clients.

You can choose between two tracks: Argonaut and IPS

In the Argonaut track, you will consume and create Argonaut compatible resources, and search FHIR servers for patient demographics, provider information and terminology using your language of choice (Java, C# or JS).

In the IPS track you will consume IPS documents and show some of the underlying resources using your language of choice (Java, C#, or JS)

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 useful specification, but in order to do real work, we need to use our desired platform to program our access to FHIR Servers.

We hope with this unit that you will grasp common concepts that apply to all clients in FHIR
and then apply them to the most commonly used platforms and languages (Java , C# , 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.