FHIR Course, Unit 2:

CRUD operations and queries on FHIR

Reading Material

Course Overview

Module I: Introduction

Introduction to FHIR

Resources

Module II: Working with FHIR

RESTful FHIR

Searching with FHIR

Module III: Advanced FHIR

Transactions

Paradigms

Messaging

Documents and CDA R2

Operations

Module IV: FHIR Conformance

Conformance Resources

Extensions

Profiles

Implementation

Table of Contents

Unit Content and Learning Objectives

This unit centers on the first interaction with a FHIR server. The basic concept of REST will be presented and we will perform different operations with resources. We are going to create, retrieve and update them, and you will learn how to query the server in various ways using different parameters.

1. REST

In this first part of the unit we will take a look at REST – REpresentational State Transfer – in the context of FHIR. The FHIR specification goes into some detail about how REST and FHIR work together, so we will draw on it for more information.

FHIR works across many different interoperability paradigms, but REST is the best established at this point in time – and by far the easiest to use (as you will see). It's the one used by companies such as Google, Amazon and Twitter to provide access to their services.

Despite the complicated-sounding name, it's actually pretty simple – in fact, it's the way the web works. The term REST was coined by Roy Fielding, one of the pioneers who defined many of the basic web protocols that we take for granted today. In a PhD dissertation, he described an 'architectural style' that took advantage of some basic web concepts in an easy-to-use way.

The basics are really simple.

REST is built on top of the HTTP protocol – the same protocol you use when requesting a web page in your browser. HTTP (short for HyperText Transfer Protocol) is simply a set of rules that both server and client (in this case a browser) use to exchange data over the web. HTTP is a 'request/response' protocol – you make a request (which can contain data), the server replies and that's it.

REST thinks in terms of ***resources***, each of which has an *identity* by which it can be manipulated. This is the URI (Uniform Resource Identifier), and uniquely specifies the location of a resource. It is specific to a particular server.

When manipulating resources, there are a stable number of *methods* that you can use. The ones that are of particular interest to FHIR include:

GET – ***retrieve*** a particular resource based on its URI (server location + id). FHIR uses the GET method both to retrieve a single resource, and to perform a search which could possibly return multiple resources.

POST – ***create*** a new resource at a specified endpoint. In REST the server will assign an id, and the combination of server location and id becomes the URI for that resource.

PUT – this is used to ***update*** an existing resource. The client that is executing the PUT needs to know the URI of the resource (which, in FHIR, means that they know the id of the resource as well as the server where it is stored). In FHIR, this will create a new version of the resource.

DELETE – ***remove*** a resource. Again, the URI must be known. It is recommended in FHIR that a copy of the resource be maintained, and potentially reachable using a version-aware read, and the history by a history request, but a plain GET should not return the deleted resource. (It will, however, return a status code telling the user that a resource was deleted.)

OPTIONS – this is a 'special' method in FHIR that is directed to the root of the server, and returns a conformance resource that informs the client what the capabilities of the server are. We will see this in the following units.

When a server responds to an HTTP REST request, it processes the call according to the *method* and its internal business logic, and then returns a response to the client. The response has a number of parts to it:

The body of the response is what the client is requesting from the server – generally in response to a GET request. Most of the other methods do not return a response.

The server always returns a status code, which informs the client what the result of the request was – did it succeed or did it fail – with some additional details about the nature of the outcome. The status code is a number, which has a well-defined meaning within the HTTP protocol. FHIR defines an operationOutcome resource, which the server can use to give the client in-depth information about any failure.

Both the request and the response contain *headers*. These provide extra information about the request or response to assist the server or client to properly process the transaction.

Examples include:

- Both request and response use the *Content-type* header to tell the other party whether a resource is in XML or JSON format.
- A client will use the *Accept* header to tell the server what format (XML/JSON) it would prefer to get back when making a GET.
- The server will use the *Location* header to tell the client the URI of a newly created resource.

There is a respectable tutorial on REST at http://rest.elkstein.org

Well, that's enough background – let's actually do something in FHIR.

Let's get a patient resource from one of the online FHIR test servers.

Paste the following command (which is actually a URI) into your browser:

http://fhirserver.hl7fundamentals.org/fhir/Patient/X10000

What this does is to instruct your browser to make a GET request to the specified URI, and display the results in your browser. You should see a JSON version of the Patient resource.

Dissecting the instruction a bit:

- HTTP:// – specified the HTTP protocol
- fhirserver.hl7fundamentals.org/fhir – the endpoint on the server that responds to FHIR requests
- Patient/ – specified that the operation was to be on the Patient resource

- X10000 – specified the patient with the id of X10000.

A couple of observations:

You need to know what the id is. It was fairly safe in this case because it is our server with some fixed resources, and unless someone else deletes that resource it should be available (try another number if that happens). You could use this technique to retrieve any type of resource from any FHIR server – provided that you knew the id – and passed any security mechanisms.

The only thing you can carry out through the browser in this way is a GET request.

The server 'wraps' the actual resource in a User Interface – you won't see this UI when making requests directly from a client (Like POSTMan) – only the XML in the middle.

2. Security & Audit

The security aspects of FHIR are still in active development, and IHE is contributing significantly to this work. FHIR itself does not define security functionality, but does depend on other services to provide that security - of which OAuth (http://oauth.net/) is one of the more important.

Using HTTPS is optional, but all production exchange of healthcare data SHOULD use SSL and additional security as appropriate. Most operations will require user authentication and some operations may depend on appropriate consent being granted.

The choice of whether to return 403 Unauthorized or 404 Not found depends upon the specific situation and specific local policies, regulations, and laws. The decision of which error to use will include consideration of whether disclosure of the existence of relevant records is considered an acceptable disclosure of PHI (personal health information) or a prohibited disclosure of PHI. Note that since a 404 does not leak information, it should be the default choice unless there is a specific reason to return a 403.

The specifics of how security concerns are addressed will vary according to the particular paradigm being used - REST, Message, Document or Service.

From an audit perspective, there are a number of resources that can be used to record activities required for auditing. These include:

- **Provenance**. This resource is used to indicate where a particular resource came from. Note that the provenance resource points to the resource that it describes and not the other way around.
- **AuditEvent** resources are equivalent to the IHE ATNA Audit Record.

Interoperability Paradigms

The intention is that the same FHIR resources should be usable in all the interoperability paradigms required in healthcare. For example, a patient resource is the same no matter how it is exchanged.

These include, other than REST:

Messages

A message is used when you want to send information from one system to another, and you expect the recipient system to update itself as required and then delete the message (other than any audit records, of course).

HL7 V2.x is all about messaging in this way.

Documents

A document is all about recording a set of patient information that applies at a 'point in time' - such as a Discharge Summary or a Referral. CDA is all about documents. FHIR handling of documents will be described in next week’s unit.

Services

A Service is also intended to be used in an online real-time way (usually), but the difference from REST is that a service can incorporate more complex workflow than the simple REST interface can provide. For example, you might use a Service in an ordering application if you wanted the service to apply basic decision support to the order, possibly modifying or rejecting it.

The use of Services is not described further in this module.

3. Manipulating a Single FHIR Patient Resource

Our next step is to manipulate a single resource in FHIR. We'll create a new one, retrieve the resource created and update it.

First task is to set up the tools we're going to need to experiment. There are at least two sets of tools you'll need to do this properly:

- Something to send and receive HTTP requests. This will to allow you to specify the content, the HTTP method and request headers, and also to view the response headers, Status Code and any response body that the server returns.
- Something to create and view XML and JSON resources. You might also want to download the FHIR schema files from the spec (look for the link that says 'FHIR Schema & Schematrons').

Depending on your environment, there are many options for both of these. To keep things simple, I'm going to use a Chrome browser extension (Postman) for making the HTTP requests and the Oxygen XML editor for the XML/JSON stuff, but you choose what works for you. I also use Notepad++ -- Google "notepad plus plus" and you'll find the website. It's a free download. If you don't have the chance to install a tool, you will be able to perform this through the campus website using our FHIR server.

Creating a New Resource

You created a Patient resource in the first week, so now we are going to send this resource to a FHIR server. It's easy – and you can do it right now. All you have to do is:

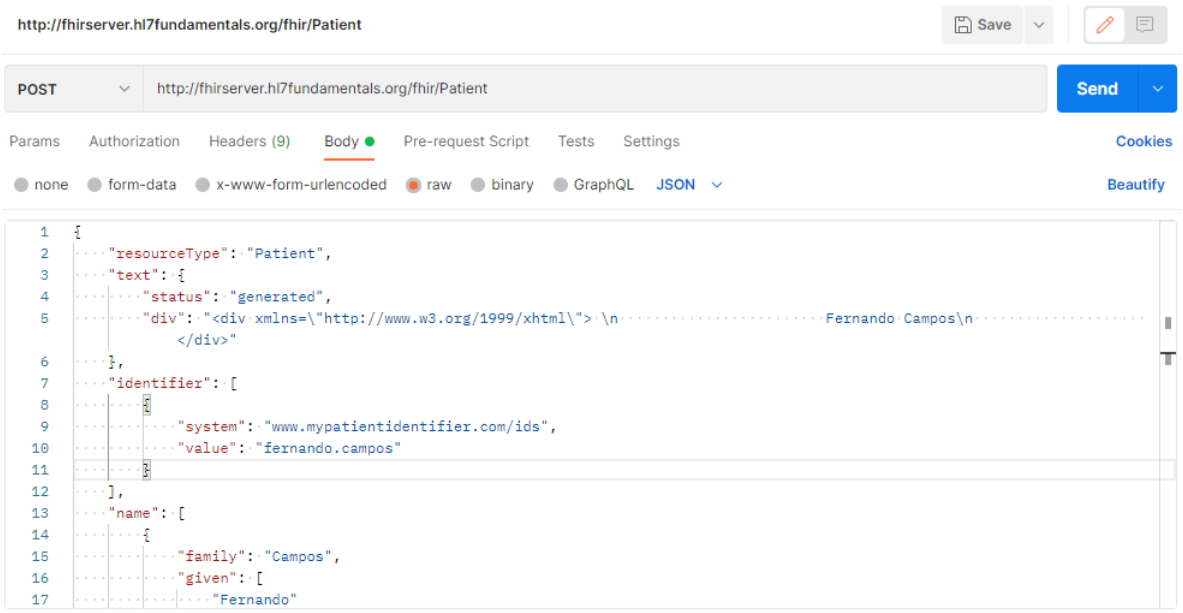
- Use the POST method

Send the request to the root for that resource on the server (you can choose any test server)– e.g.

http://fhirserver.hl7fundamentals.org/fhir/Patient in the case of our course server – rather than to a URI (after all, you don't know what ID the server is going to assign to the new resource yet).

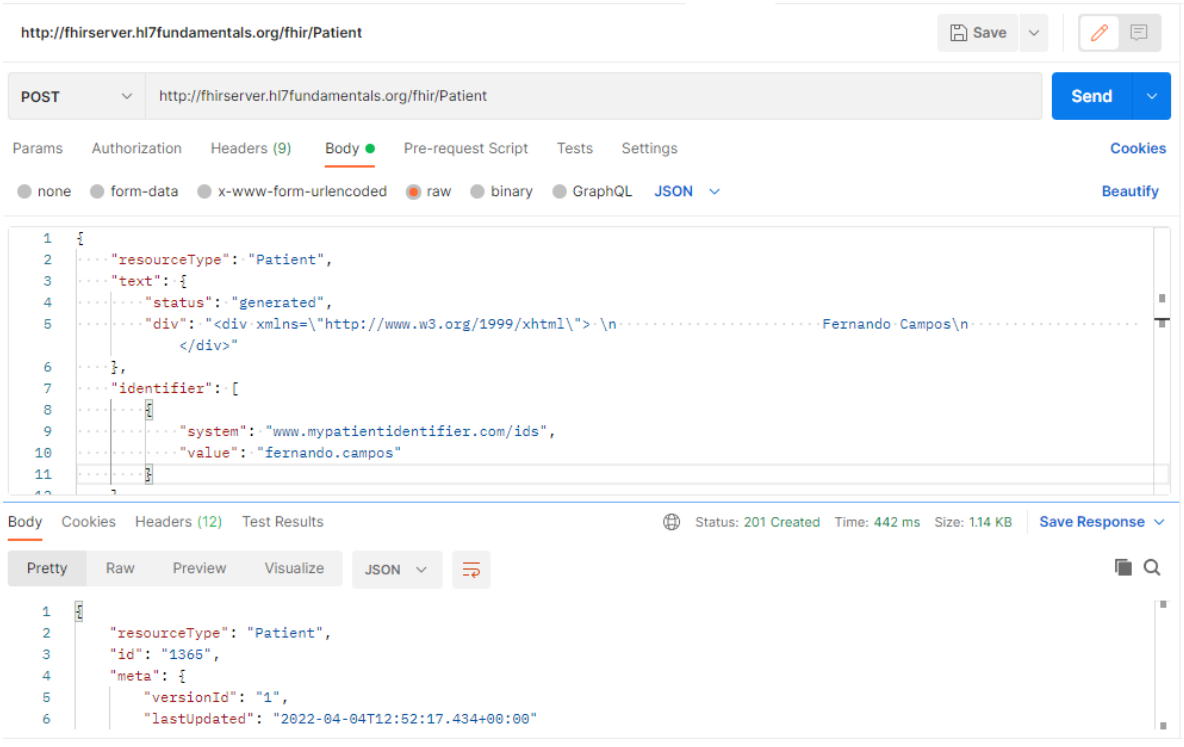
All you have to do is enter the URL in POSTMan, select the method to POST, set the content-type, paste your resource and press <Send>.

I'll show how I did it with our test server.



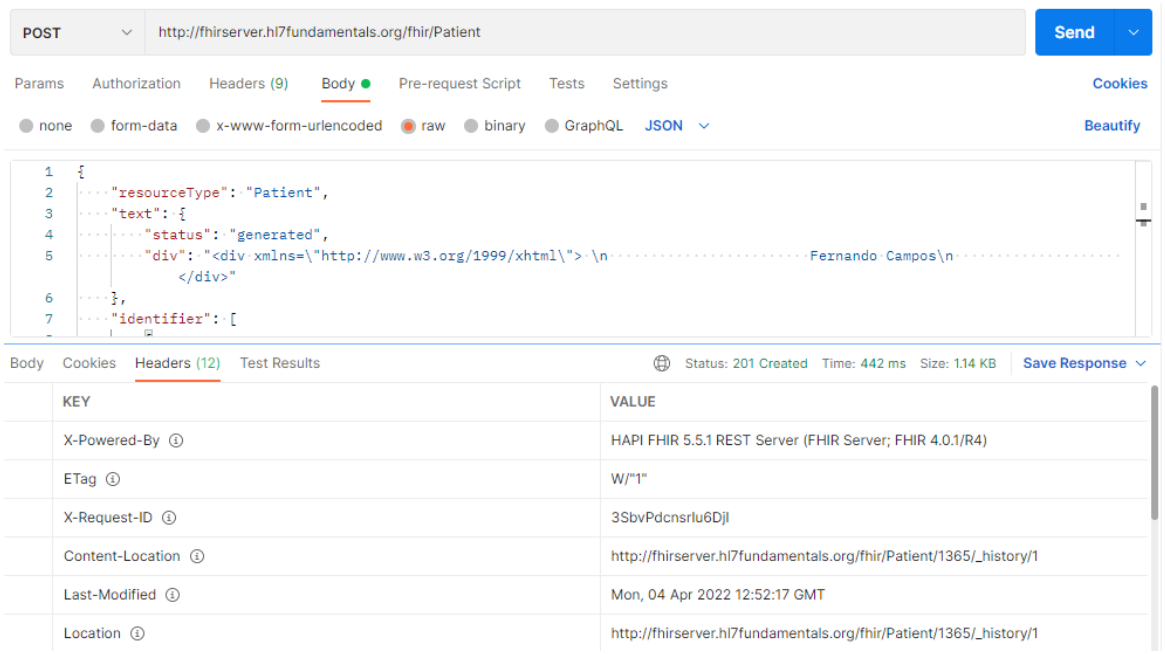
If your resource is in XML format, you will need to change the header Content-type to application/fhir+xml instead of application/fhir+json.

The server will process the request and return a response. This time the status code will be 201, indicating that a new resource was created and an ID will be assigned to the resource.



Remember this ID we will use to retrieve the resource.

The actual URI for the resource will be in the Content-Location header (and it will, of course, be a version-specific URI).



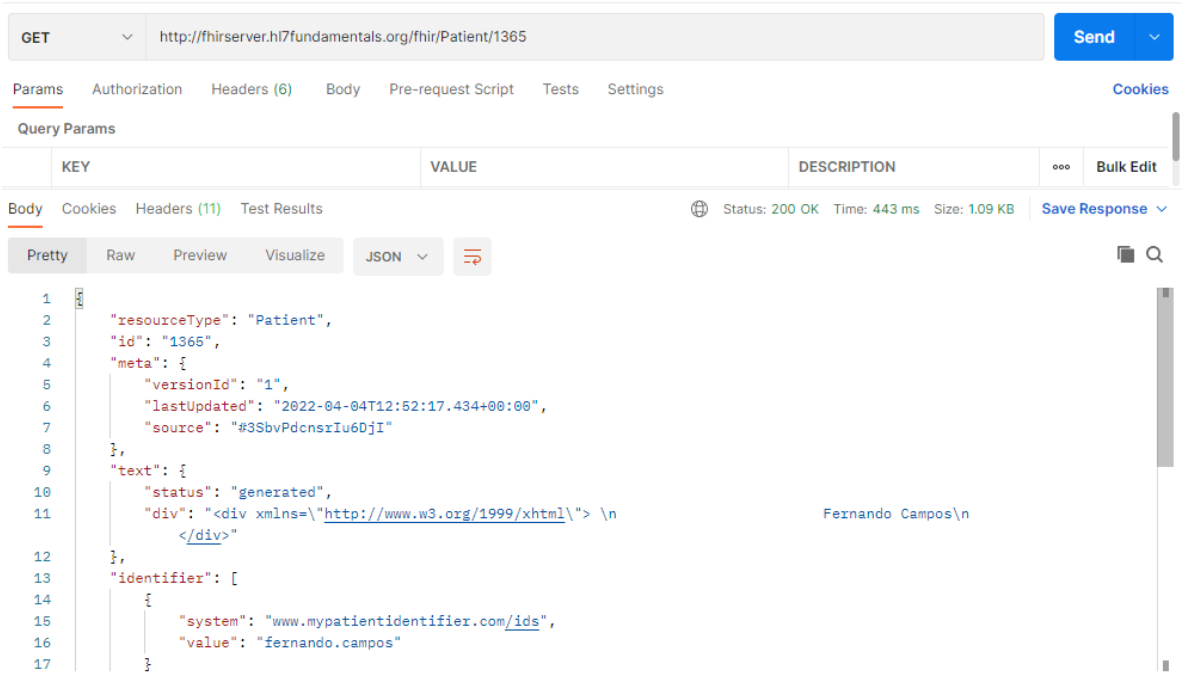
Can a client rather than the server create the ID?

It is certainly possible for the client to assign the ID – in FHIR, if a client PUTs a resource to a URI (i.e. includes the ID) and there is no resource already there, then the server will create the resource using that ID (it returns a 201 status to indicate that this has happened). The issue with this, of course, is potential ID collisions.

Reading an Existing Resource

To read an existing resource when you know its URI (i.e. the server location and ID on that server) you use an HTTP GET request. A test server will be used for this example, which is a public server that anyone can change, so the details may be different for you.

Here's what I got using Postman to retrieve the Patient I created.



Note that:

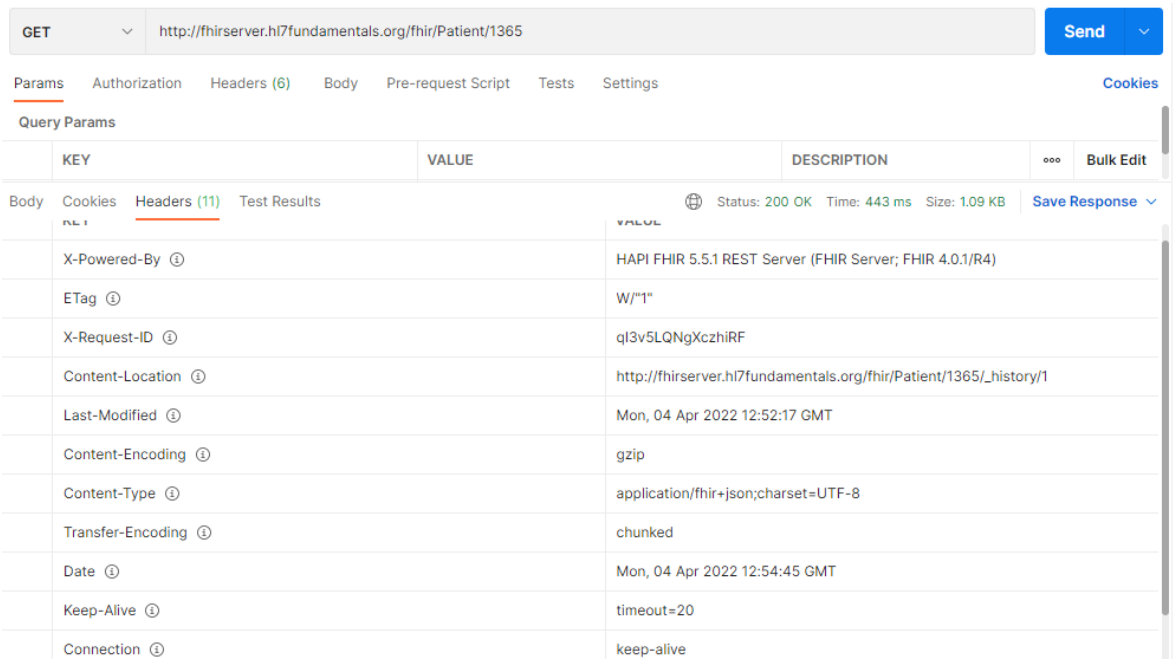
- The full URI to that resource on the server is http://fhirserver.hl7fundamentals.org/fhir/Patient/1365

- You could use the URI at the Content-Location header when you created the resource http://fhirserver.hl7fundamentals.org/fhir/Patient/1365/_history/1 - try it with our server you too!

- I asked the server to return the resource in the JSON format, by setting a request header – *Accept* – to 'application/fhir+json. You could have set the *Accept* header to 'application/fhir+xml to get a XML formatted resource – try it!

- The Status Code was 200 – which means everything worked OK.

In Postman, I can also look at the response headers – which the server sets – as shown below:



Notes:

- The content-type header tells us that this is a JSON format.
- There are a number of other useful headers that we won't go into right now.

Updating a Resource

Let's update this resource. What we'll do is copy the resource we just retrieved into our XML editor, make some changes, paste them back into Postman and **PUT** them to the server. Optionally, we'll validate the updated resource using the FHIR schema before we send it to the server.

Step 1: Copy the resource you just downloaded.

Step 2: Paste the resource into your editor.

Step 3: Change the file – for example, you could add a new name just after the existing name, like this:

```
...
    "name": [
        {
            "family": "Campos",
            "given": [
                "Fernando on FHIR"
            ]
        },...
```

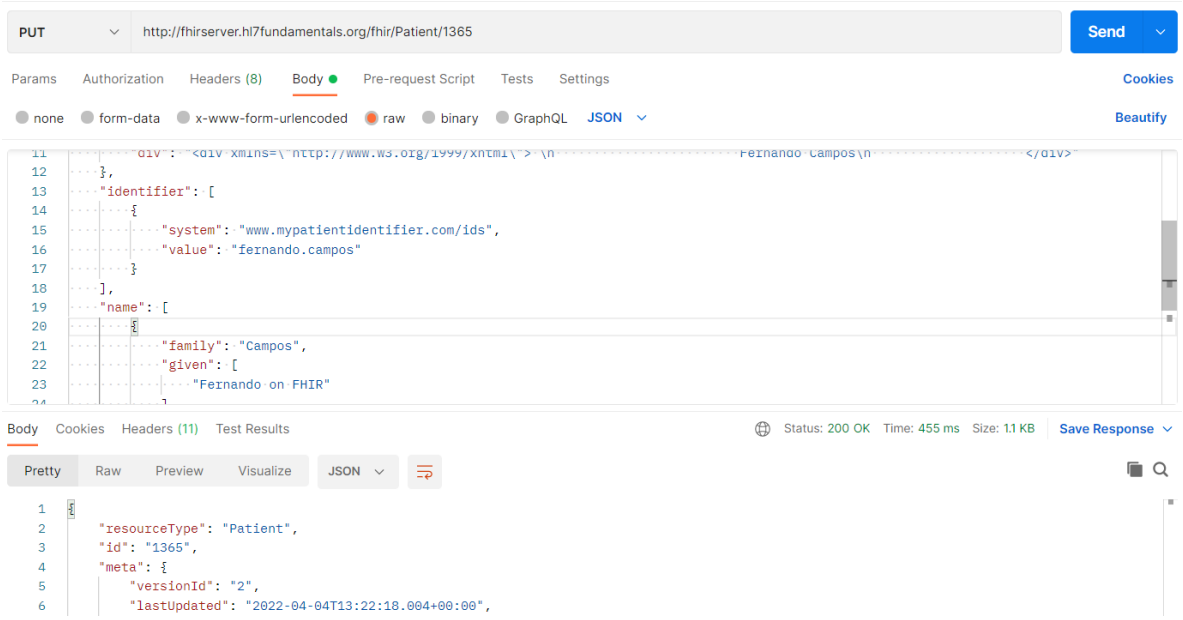
If you want, you can check that this is valid against the FHIR schema by associating the updated file with the correct XML schema (it will be Patient.xsd in whatever folder you downloaded the schema to) and clicking the 'validate' button in your editor.

Step 4: Copy the changed resource.

Step 5: In Postman, change the GET to a PUT and paste the updated resource into the body of the request (the raw tab is the best).

Step 6: Press the Send button.

If all goes well, the server should accept the changed resource, and return a response with a status code of 200. The Content-Location header should show that the version has been incremented. Here's what I got:



Notes:

- The spec does not require that version numbers be sequential – although this server does make them sequential. Others use a different scheme – all that matters is that a new version of a resource gets a new version number.
- We used PUT and not POST because we knew the full URI (including the resource ID). POST would have indicated that we wanted to create a new resource.
- If you PUT to a URI and there is no existing resource at that URI, it will still be saved, but the status code that is returned will be 201 rather than 200.

And before we move on from the topic of updates – and versions:

- You can always retrieve a specific version of a resource using the version-specific URI that was returned when the resource was updated – e.g. http://fhirserver.hl7fundamentals.org/fhir/Patient/1365/_history/2

- You can get a list of all the versions that exist for a resource by appending the term 'history' to the URI of the resource – e.g. http://fhirserver.hl7fundamentals.org/fhir/Patient/1365/_history
- Note that this will return a list of resources in a bundle – we will cover bundles when we perform queries.

TRY IT

Do play around with reading, creating and updating Patient resources. You will need to be trained to perform the assignment! You can work with the resource you have created in your editor (and don't forget to use the schema validation so you know you have a valid resource). Read the documentation for the Patient resource to find out what properties you can add, and don't forget to look at the examples in the spec (in a tab at the top of the page).

Deleting a Resource

We are not going to practice how to remove a resource, but you have to use the DELETE verb, e.g.:

```
DELETE <host>/<resourceType>/<id>
```

Assuming the delete was successful (i.e. there was no business process that prevented it occurring), then the server will return a status code of 204 (no content). If the delete cannot occur, then the returned status code is 405 (method not allowed), 409 (conflict) or 404 (not found).

A successful delete instruction means that the server will no longer return a resource to a simple GET request for that resource. If such a request is made, then the server will return a status code of 410 (gone) - note that this is different from requesting a resource that has never existed, which would return a 404.

The previous versions of the resource are not removed, and can be retrieved using a version specific request.

Transaction Batch Updates

The transaction update facility allows a client to submit a number of different resources in a single bundle. There are two main purposes:

- To support the notion of a transaction (since the transaction processing must succeed or fail as a whole)
- To support push-based publication-subscription - e.g. where different servers are being synchronized.

From a server perspective, transaction processing is hard, especially with regard to id management, but from a client perspective it is easy - just create a bundle with the given resources and POST it to the server root.

The server processes the transaction as if each resource had been submitted separately (though it does need to treat the transaction as a transaction and be able to roll back a failed update).

After processing, the updated transaction is returned to the client (and there may be some differences between the submitted transaction and the processed transaction - especially if id's were assigned by the server).

Note that although the transaction processing uses a bundle to contain the resources to be processed, there are a couple of important differences:

- The order of resources in the transaction must not be significant.
- A particular resource can only occur once in the transaction.

If you do intend to use transaction processing, read the specification carefully.

4. FHIR Searching

In this part we are going to focus on searching using the REST paradigm within FHIR. This is a large topic, and we certainly won't cover it in detail, but the intention is to provide enough information for you to get started. (For full details on searching or queries, refer to the specification.)

We're going to focus on searching for a single type of resource – e.g. find a patient – how to specify the search parameters and how FHIR returns the results. Then you should practice this on a Public server, and you also have to complete the activity on the course server.

Each resource has a list of predefined search parameters that are defined in the specification for that resource. These are at the bottom of the page that describes each resource – for example, here are the patient search parameters.

8.1.12 Search Parameters

Search parameters for this resource. The common parameters also apply. See Searching for more information about searching in REST, messaging, and services.

Name	Type	Description	Expression	In Common
active TU	token	Whether the patient record is active	Patient.active	
address TU	string	A server defined search that may match any of the string fields in the Address, including line, city, district, state, country, postalCode, and/or text	Patient.address	3 Resources
address-city TU	string	A city specified in an address	Patient.address.city	3 Resources
address-country TU	string	A country specified in an address	Patient.address.country	3 Resources
address-postalcode TU	string	A postalCode specified in an address	Patient.address.postalCode	3 Resources
address-state TU	string	A state specified in an address	Patient.address.state	3 Resources
address-use TU	token	A use code specified in an address	Patient.address.use	3 Resources
birthdate TU	date	The patient's date of birth	Patient.birthDate	2 Resources
death-date TU	date	The date of death has been provided and satisfies this search value	(Patient.deceased as dateTime)	
deceased TU	token	This patient has been marked as deceased, or as a death date entered	Patient.deceased.exists() and Patient.deceased != false	
email TU	token	A value in an email contact	Patient.telecom.where(system='email')	4 Resources
family TU	string	A portion of the family name of the patient	Patient.name.family	1 Resources
gender TU	token	Gender of the patient	Patient.gender	3 Resources
general-practitioner TU	reference	Patient's nominated general practitioner, not the organization that manages the record	Patient.generalPractitioner (Practitioner, Organization, PractitionerRole)	
given TU	string	A portion of the given name of the patient	Patient.name.given	1 Resources
identifier TU	token	A patient identifier	Patient.identifier	
language TU	token	Language code (irrespective of use value)	Patient.communication.language	
link TU	reference	All patients linked to the given patient	Patient.link.other (Patient, RelatedPerson)	
name TU	string	A server defined search that may match any of the string fields in the HumanName, including family, give, prefix, suffix, suffix, and/or text	Patient.name	
organization TU	reference	The organization that is the custodian of the patient record	Patient.managingOrganization (Organization)	
phone TU	token	A value in a phone contact	Patient.telecom.where(system='phone')	4 Resources
phonetic TU	string	A portion of either family or given name using some kind of phonetic matching algorithm	Patient.name	3 Resources
telecom TU	token	The value in any kind of telecom details of the patient	Patient.telecom	4 Resources

Note that FHIR servers are not required to support all these search parameters, and servers are also able to define their own searches. This is simply a list of the most common or obvious parameters, as compiled by the committee that defined the resource.

A FHIR server can indicate in its conformance resource which parameters it supports for each resource.

The format for this basic searching uses the GET method as follows:

```
GET [base]/[type]?[parameters] &_format=[mime-type]
```

or

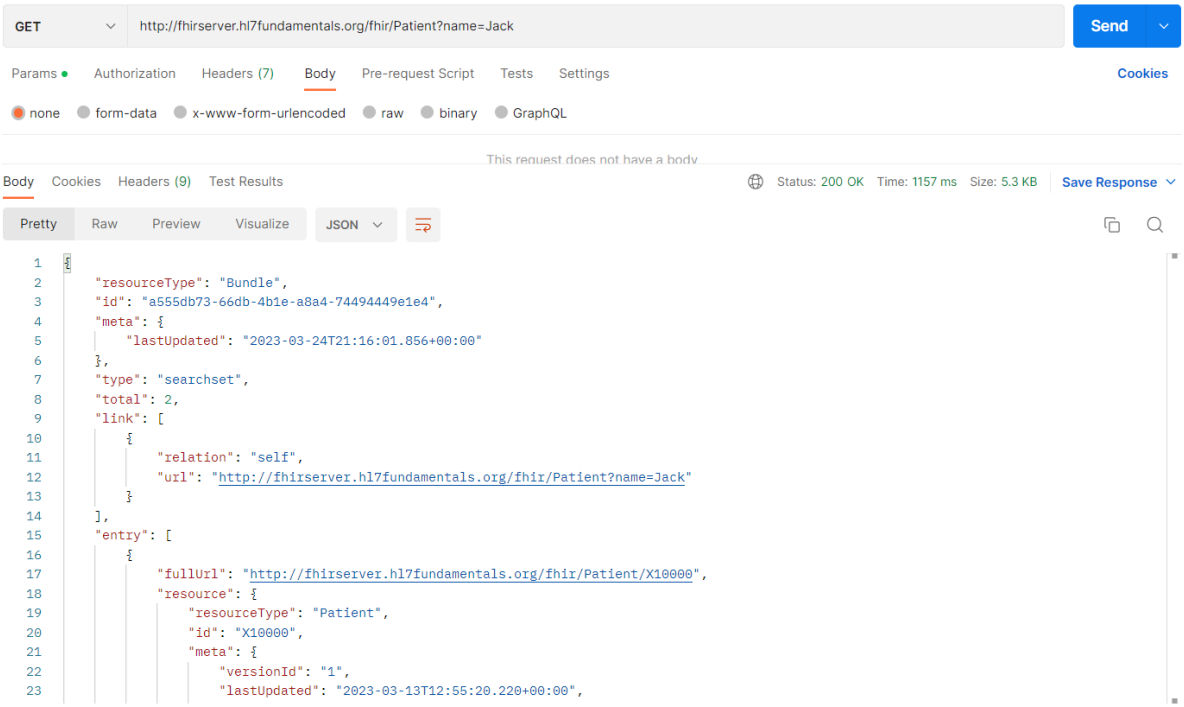
```
GET [base]/[type]/_search?[parameters] &_format=[mime-type]
```

Where:

- Base is the base URL to the server
- Type is the resource type (like 'Patient')
- Parameters are the search parameters you wish to use (and there can be more than one)
- Format is the format (XML or JSON) you want the results in. (HTTP headers are the more elegant way to specify this, but it might not always be possible to easily set the headers, which is why this format is supported. In fact, it also applies to simple resource retrieval as well.)

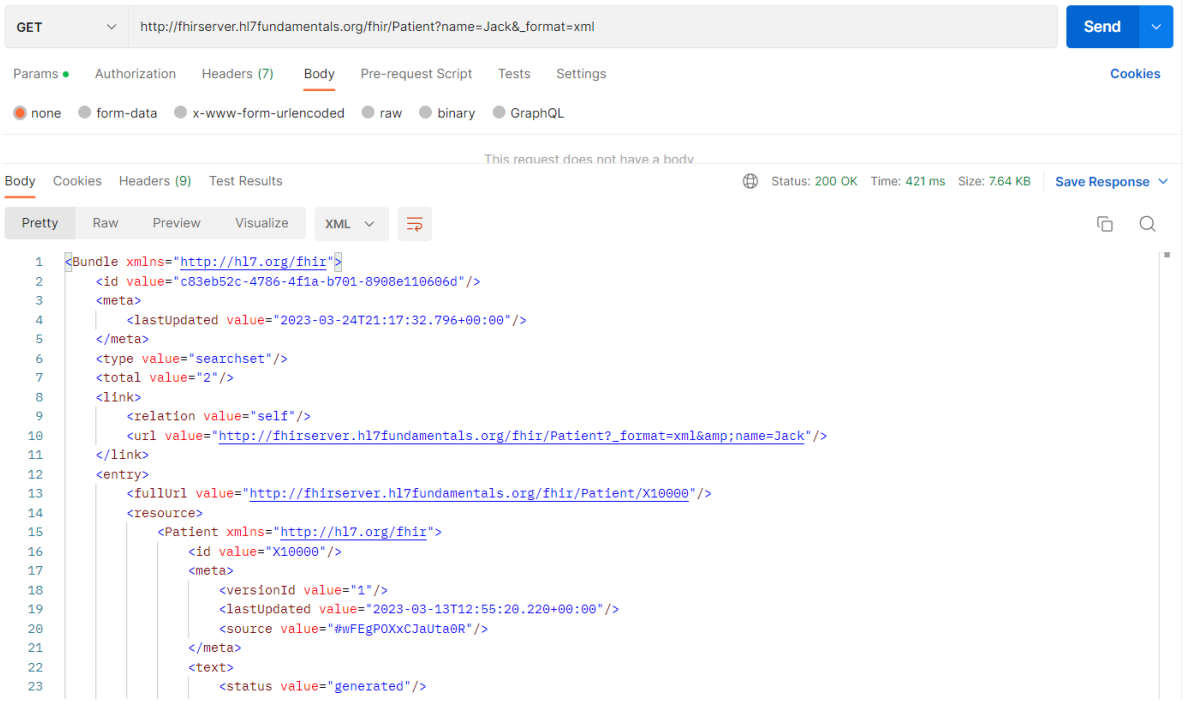
The parameters are expressed in name/value pairs, e.g.

http://fhirserver.hl7fundamentals.org/fhir/Patient?name=Jack would return all patients whose name contains the word 'jack'



http://fhirserver.hl7fundamentals.org/fhir/Patient?name=jack&_format=xml

is the same search, but specifying that the result should be in a JSON format rather than the default XML format.



Each resource parameter has a *type* that defines how it behaves – e.g. whether it is text, a number or part of a code set (a token). In most cases this is reasonably straightforward, although there are subtleties in some cases. The one that causes the most difficulty at first is where one of the parameters is a resource reference – e.g. you want all the conditions for a particular patient. (In fact, this particular scenario has a specific solution: the compartment.)

The following search:

http://fhirserver.hl7fundamentals.org/fhir/Observation?subject=X12984 will return all Observations where the ID of the subject (which is a patient) is X12984 (the assumption is that you have previously performed a search on the patient to locate that ID). It is possible to get more complicated than that – for example, you can chain query parameters to get, say, all conditions for patients whose name is Sam. Go look at the spec if that functionality is important in your use case. (Note: the format of this type of query has changed recently and not all servers may support it yet.)

Retrieve a Resource

To retrieve a resource based on its id, issue a GET request based on the following pattern

```
GET <host>/<resourceType>/<id>
```

Where:

- <host> is the name of the server – e.g. http://fhirserver.hl7fundamentals.org/fhir
- <resource> is the name of the resource – e.g. 'Patient' or 'Observation'
- <id> is the id of the resource you are after.

Thus:

GET http://fhirserver.hl7fundamentals.org/fhir/Patient/X10000 will return the patient resource with the id of X10000 from the test server. The response status is important:

- 200 indicates that the resource was found
- 410 means there was a resource with this id, but it has been deleted
- 404 indicates 'not found'.

The server will also return a number of response headers. These include:

- **Content-Location** will be the full location to the resource, including any version - i.e. a version specific URL
- **Content-Type** will be the format of the resource - XML or JSON

But there will be others.

You can specify the format in which the server should return the resource in two ways:

- In the request, set the request header '**Accept'** to *application/fhir+json* or *application/fhir+xml* to specify JSON or XML respectively. This is the preferred approach.
- In the request URL, use the "_format" parameter to specify the format:

The latter option is provided as a convenience for those clients that cannot (or will not) set request headers.

You will generally get the id of a resource as a result of a search operation (see below).

A GET is considered 'idempotent' - you can issue it any number of times without affecting the server.

Standard Parameters

Parameters for all resources

The following parameters apply to all resources:

`_id`

`_lastUpdated`

`_tag`

`_profile`

`_security`

`_text`

`_content`

`_list`

`_has`

`_type`

In addition, there is a special search parameters _query and _filter that allow for an alternative method of searching, and the parameters _format and _pretty defined for all interactions.The search parameter `_id` refers to the logical id of the resource, and can be used when the search context specifies a resource type:

```
GET [base]/Patient?_id=X10000
```

This search finds the patient resource with the given id (there can only be one resource for a given id). Functionally, this is equivalent to a simple read operation:

```
GET [base]/Patient/X10000
```

However, the search with parameter `_id` returns a bundle with the requested resource, instead of just the resource itself.

The basic search mechanism is flexible, and easy to implement for simple cases, but is limited in its ability to express combination queries. To complement this mechanism, the "_filter" search expression parameter can be used.

For example, "Find all the observations for patient with a name including peter that have a LOINC code 1234-5":

```
GET [base]/Observation?code=http://loinc.org|1234-5&subject.name=peter
```

So we've described how to find resources based on matching their properties to some value and how to find resources based on their link to some other resource, but we haven't talked about the format that the results are returned in. It can't just be the resource, as there will likely be more than one that matches – it has to be a set of some sort.

FHIR uses the Atom format to return the results of any search – regardless of how many resources matched the search – and this is what we call **a bundle**.

Bundles are, in fact, used in a number of parts of FHIR – they form the basis of FHIR Documents and FHIR Messages, as well as the ability to batch multiple commands into a single transaction -- but let's stay focused on search for now.

So, to summarize, a search in RESTful FHIR is quite simple:

- Send a GET request with the desired parameters as name/value pairs in the URL

- Iterate over the <entry> elements in the response to get the results (or just use an XSLT transform displaying the title & summary elements if all you are doing is displaying it to the user or similar simple processing).

Additional parameters can be included which may provide extra functionality on top of this base read equivalence (e.g. `_include`).

The search parameter `_lastUpdated` can be used to select resources based on the last time they were changed:

```
GET [base]/Observation?_lastUpdated=>2010-10-01
```

Search Parameter Types

Each search parameter is defined by a type that specifies how the search parameter behaves. These are the defined parameter types:

- **Number**
  - Search parameter SHALL be a number (a whole number or a decimal).
- **Date**
  - Search parameter is on a date/time. The date format is the standard XML format, though other formats may be supported.
- **String**
  - Search parameter is a simple string, like a name part. Search is case-insensitive and accent-insensitive. May match just the start of a string. String parameters may contain spaces.
- **Token**
  - Search parameter on a coded element or identifier. May be used to search through the text, displayname, code and code/codesystem (for codes) and label, system and key (for identifier). Its value is either a string or a pair of namespace and value, separated by a "|", depending on the modifier used.
- **Reference**
  - A reference to another resource.

- **Composite**
    - A composite search parameter that combines a search on two values together.
- **Quantity**
    - A search parameter that searches on a quantity.
- **Uri**
    - A search parameter that searches on a URI (RFC 3986).

The search parameters can also append "modifiers" that control their behavior. The kinds of modifiers available are dependent on the type of parameter being modified.

Modifiers

Parameters are defined per resource. Parameter names may specify a modifier as a suffix. The modifiers are separated from the parameter name by a colon.

Modifiers are:

- **For all parameters** (except combination): **missing**; e.g. gender:missing=true (or false). Searching for gender:missing=true will return all the resources that don't have a value for the gender parameter (which usually equates to not having the relevant element in the resource). Searching for gender:missing=false will return all the resources that have a value for the gender parameter.
- **For string**: **:exact** (the match needs to be exact, no partial matches, case sensitive and accent-sensitive), or :contains (case insensitive and accent-insensitive, partial match at start or end), instead of the default behavior (case insensitive and accent-insensitive, partial matches at the end of the string).
- **For token**: **:text** (the match does a partial searches on the text portion of a CodeableConcept or the display portion of a Coding), instead of the default search which uses codes. Other defined modifiers are :in, :below, :above and :not-in which are described below.
- **For reference**: **:[type]** where [type] is the name of a type of resource
- **For uri**: **:below**, :above indicate that instead of an exact match, either the search term left matches the value, or vice-versa.

If you send a search request that contains or is suffixed by a modifier that the server does not support for that parameter, the FHIR server will reject it. For example, if the server supports the name search param, but not the :exact modifier on the name, it should reject a search with the parameter name:exact=Bill, using an HTTP 400 error with an OperationOutcome with a clear error message.

Prefixes

For the ordered parameter types of **number**, **date**, and **quantity**, a prefix to the parameter value may be used to control the nature of the matching.

To avoid URL escaping and visual confusion, the following prefixes are used:

eq	the value for the parameter in the resource is equal to the provided value	the range of the search value fully contains the range of the target value
ne	the value for the parameter in the resource is not equal to the provided value	the range of the search value does not fully contain the range of the target value
gt	the value for the parameter in the resource is greater than the provided value	the range above the search value intersects (i.e. overlaps) with the range of the target value
lt	the value for the parameter in the resource is less than the provided value	the range below the search value intersects (i.e. overlaps) with the range of the target value
ge	the value for the parameter in the resource is greater or equal to the provided value	the range above the search value intersects (i.e. overlaps) with the range of the target value, or the range of the search value fully contains the range of the target value
le	the value for the parameter in the resource is less or equal to the provided value	the range below the search value intersects (i.e. overlaps) with the range of the target value or the range of the search value fully contains the range of the target value
sa	the value for the parameter in the resource starts after the provided value	the range of the search value does not overlap with the range of the target value, and the range below the search value contains the range of the target value
eb	the value for the parameter in the resource ends before the provided value	the range of the search value does overlap not with the range of the target value, and the range above the search value contains the range of the target value
ap	the value for the parameter in the resource is approximately the same to the provided value. Note that the recommended value for the approximation is 10% of the stated value (or for a date, 10% of the gap between now and the date), but systems may choose other values where appropriate	the range of the search value overlaps with the range of the target value

If no prefix is present, the prefix eq is assumed. Note that the way search parameters operate is not the same as the way the operations on two numbers work in a mathematical sense. sa (starts-after) and eb (ends-before) are not used with integer values.

Number

Searching on a simple numerical value in a resource. Examples:

`[parameter]=100`	Values that equal 100, to 3 significant figures precision, so range [99.5 ... 100.5)
`[parameter]=100.00`	Values that equal 100, to 5 significant figures precision, so range [99.995 ... 100.005). Whole numbers also equal 100.00, but not 100.01
`[parameter]=lt100`	Values that are less than 100
`[parameter]=le100`	Values that are less or equal to 100
`[parameter]=gt100`	Values that are greater than 100
`[parameter]=ge100`	Values that are greater or equal to 100
`[parameter]=ne100`	Values that are not equal to 100

Note: Uncertainty does not factor in evaluations. The precision of the numbers is considered arbitrarily high. (The way search parameters operate in resources is not the same as whether two numbers are equal to each other in a mathematical sense.)

Here are some example searches:

Search	Description
`GET [base]/Encounter?length=gt20`	Search for all the encounters longer than 20 days
`GET [base]/ImmunizationRecommendation?deo-number=2`	Search for any immunization recommendation recommending a second dose

Date

A date parameter searches on a date/time or period. As is usual for date/time-related functionality, while the concepts are relatively straightforward, there are a number of subtleties involved in ensuring consistent behavior.

The date parameter format is `yyyy-mm-ddThh:mm:ss[Z|(+|-)hh:mm]` (the standard XML format).

Technically, this is any of the **date**, **dateTime**, and **instant** data types; e.g. any degree of precision can be provided, but it SHALL be populated from the left (e.g. can't specify a month without a

year), except that the minutes SHALL be present if an hour is present, and you SHOULD provide a time zone if the time part is present. Note: Time can consist of hours and minutes with no seconds, unlike the XML Schema dateTime type. Some user agents may escape the : characters in the URL, and servers SHALL handle this correctly.

Date parameters may be used with the following data types:

date	The range of the value is the day, month, or year as specified
dateTime	The range of the value as defined above; e.g. For example, the date 2013-01-10 specifies all the time from 00:00 on 10-Jan 2013 to immediately before 00:00 on 11-Jan 2013
instant	An instant is considered a fixed point in time with an interval smaller than the precision of the system, i.e. an interval with an effective width of 0
Period	Explicit, though the upper or lower bound may not actually be specified in resources.
Timing	The specified scheduling details are ignored and only the outer limits matter. For instance, a schedule that specifies every second day between 31-Jan 2013 and 24-Mar 2013 includes 1-Feb 2013, even though that is on an odd day that is not specified by the period. This is to keep the server load processing queries reasonable.

Implicitly, a missing lower boundary is "less than" any actual date. A missing upper boundary is "greater than" any actual date. The use of the prefixes:

`[parameter]=eq2013-01-14`	Matches 2013-01-14T00:00 (obviously) and also 2013-01-14T10:00 but not 2013-01-15T00:00
`[parameter]=ne2013-01-14`	Matches 2013-01-15T00:00 but not 2013-01-14T00:00 or 2013-01-14T10:00
`[parameter]=lt2013-01-14T10:00`	Includes the time 2013-01-14, because it includes the part of 14-Jan 2013 before 10am
`[parameter]=gt2013-01-14T10:00`	Includes the time 2013-01-14, because it includes the part of 14-Jan 2013 after 10am
`[parameter]=ge2013-03-14`	Includes a period "from 21-Jan 2013 onwards", because that period may include times after 14-Mar 2013

Other notes:

- When the date parameter is not fully specified, matches against it are based on the behavior of intervals, where:

- Dates with only the year specified are equivalent to an interval that starts at the first instant of Jan. 1 to the last instant of Dec. 31, e.g. 2000 is equivalent to an interval of [2000-01-01T00:00, 2000-12-31T23:59].
- Dates with the year and month are equivalent to an interval that starts at the first instant of the first day of the month and ends on the last instant of the last day of the month, e.g. 2000-04 is equivalent to an interval of [2000-04-01T00:00, 2000-04-30T23:59].
- Where possible, the system should correct for time zones when performing queries. Dates do not have time zones, and time zones should not be considered. Where both search parameters and resource element date times do not have time zones, the server's local time zone should be assumed.

To search for all the procedures in a patient compartment that occurred over a 2-year period:

```
GET [base]/Patient/23/Procedure?date=ge2010-01-01&date=le2011-12-31
```

String

For a simple string search, a string parameter serves as the input for a case- and accent-insensitive search against sequences of characters. By default, a field matches a string query if the value of the field equals or starts with the supplied parameter value, after both have been normalized by case and accent. The `:contains` modifier returns results that include the supplied parameter value anywhere within the field being searched. The `:exact` modifier returns results that match the entire supplied parameter, including casing and accents.

Examples:

`[base]/Patient?name=eve`	Any patients with a name containing "eve" at the start of the name. This would include patients with the name "Eve", "Evelyn".
`[base]/Patient?name:contains=eve`	Any patients with a name containing "eve" at any position. This would include patients with the name "Eve", "Evelyn", and also "Severine".
`[base]/Patient?name:exact=Eve`	Any patients with a name that is exactly "Eve". Note: This would not include patients with the name "eve" or "EVE".

It is at the discretion of the server whether to pre-process names, addresses, and contact details to remove separator characters prior to matching in order to ensure more consistent behavior. For example, a server might remove all spaces and non-numeral characters from phone numbers. What is most appropriate varies depending on culture and context.

Example searches:

Search	Description
`GET [base]/Patient?identifier=http://hl7fundamentals.org/patient|12345`	Search for all the patients with an identifier with key = "12345" in the system " http://hl7fundamentals /patient"
`GET [base]/Patient?gender=female`	Search for any patient with a gender that has the code "female"
`GET [base]/Patient?gender:not=female`	Search for any patient with a gender that does not have the code "female"
`GET [base]/Patient?active=true`	Search for any patients that are active
`GET [base]/Condition?code=http://hl7fundamentals.org/conditions/codes|ha543`	Search for any condition with a code "ha543" in the code system " http://hl7fundamentals.org /conditions/codes"
`GET [base]/Condition?code=ha542`	Search for any condition with a code "ha542". Note that there is not often any useful overlap in literal symbols between code systems, so the previous example is generally preferred
`GET [base]/Condition?code:text=headache`	Search for any Condition with a code that has a text "headache" associated with it (either in the text, or a display)
`GET [base]/Condition?code:below=126851005`	Search for any condition that is subsumed by the SNOMED CT Code "Neoplasm of liver".

Searching Across Resources

Cross-resource searching is a bit complex at present.

A common scenario is to 'de-reference' a resourceReference datatype - for example, suppose you want all the conditions for a particular patient.

Examining the Condition resource shows that the patient for a condition is given by the 'subject' element, which is a resource reference - i.e. it refers to the Patient resource with a given id. So, if the patient id is 'example', then:

/Patient/example would return the Patient resource

/Condition?subject._id=example would return all the problems where the subject was the resource whose id was 'example'

Note that this will only work if all the resources are on the same server. The situation gets a lot more complicated if there are Condition resources on different servers, or where the Condition resource on one server has a resourceReference link to a patient on another server.

Retrieve the History of a Specific Resource

As described above, FHIR has the concept of resource versions. A server is not obliged to provide versioning functionality, but if it does, then there is a specified way of using it.

The format for this request is:

```
GET <host>/<resourceType>/<id>/_history
```

This will return a bundle containing all the previous versions for the resource.

(As an aside, it is also possible for a server to return history at a higher level - either all the changes for a specific resource type, or all resources. This is intended for use in server synchronization scenarios.)

Retrieve a Specific Version of a Resource

You can also return just the specific version (assuming you have both the resource id and the version id).

The format for this request is:

```
GET <host>/<resourceType>/<id>/_history/<versionId>
```

The result will be the specified version of the resource (unless it was deleted, in which case the server will return a 410 status code).

Paging

If servers provide paging for the results of a search or history interaction, they SHALL conform to this method for sending continuation links to the client when returning a bundle (e.g. with `history` and `search`). If the server does not do this, there is no way to continue paging.

An example:

```
<Bundle xmlns="http://hl7.org/fhir">
  <link>
    <relation value="self">
    <url value="http://example.org/Patient?name=peter&stateid=23&page=3"/>
  </link>
  <!-- 4 links for navigation in the search. All of these are optional, but recommended -
->


  <link>
```

```
    <relation value="first"/>
    <url value="http://example.org/Patient?name=peter&stateid=23&page=1"/>
  </link>
  <link>
    <relation value="previous"/>
    <url value="http://example.org/Patient?name=peter&stateid=23&page=2"/>
  </link>
  <link>
    <relation value="next"/>
    <url value="http://example.org/Patient?name=peter&stateid=23&page=4"/>
  </link>
  <link>
    <relation value="last"/>
    <url value="http://example.org/Patient?name=peter&stateid=23&page=26"/>
  </link>

  <!-- then the search results... -->
</Bundle>
```

The server need not use a stateful paging method as shown in this example - it is at the discretion of the server how to best ensure that the continuation retains integrity in the context of ongoing changes to the resources. An alternative approach is to use version-specific references to the records on the boundaries, but this is subject to continuity failures when records are updated.

A server MAY inform the client of the total number of resources returned by the interaction for which the results are paged using the Bundle.total.

Note that for search, where _include can be used to return additional related resources, the total number of resources in the feed may exceed the number indicated in totalResults.

Unit Summary and Conclusion

In this unit we learned how to interact with a FHIR server. In the next units, we will go into more detail on profiles and conformance as well as on the architecture of FHIR and implementation considerations.