Article
Lucas Enard · May 13, 2022 8m read

Open Exchange

A simple example of a Fhir client in java

1. Fhir-client-java

This is a simple fhir client in java to practice with fhir resources and CRUD requests to a fhir server.

Note that for the most part auto-completion is activated.

GitHub

2. Prerequisites

Make sure you have git and Docker desktop installed.

Already installed in the container :

Hapi Fhir model and client

3. Installation

3.1. Installation for development

Clone/git pull the repo into any local directory e.g. like it is shown below:

```
git clone https://github.com/LucasEnard/fhir-client-java.git
```

Open the terminal in this directory and run:

```
docker build .
```

3.2. Management Portal and VSCode

This repository is ready for VS Code.

Open the locally-cloned fhir-client-java folder in VS Code.

If prompted (bottom right corner), install the recommended extensions.

3.3. Having the folder open inside the container

You can be *inside* the container before coding if you wish.

For this, docker must be on before opening VSCode.

Then, inside VSCode, when prompted (in the right bottom corner), reopen the folder inside the container so you will be able to use the python components within it.

The first time you do this it may take several minutes while the container is readied.

If you don't have this option, you can click in the bottom left corner and press reopen in container then select From Dockerfile

More information here

By opening the folder remote you enable VS Code and any terminals you open within it to use the java components within the container.

4. FHIR server

To complete this walktrough you will need a FHIR server.

You can either use your own or go to InterSystems free FHIR trial and follow the next few steps to set it up.

Using our free trial, just create an account and start a deployement, then in the Overview tab you will get acces to an endpoint like https://fhir.000000000.static-test-account.isccloud.io that we will use later.

Then, by going to the Credentials tab, create an api key and save it somewhere.

Now you are all done, you have you own fhir server holding up to 20GB of data with a 8GB memory.

5. Walkthrough

Complete walkthrough of the client situated at src/java/test/Client.java.

The code is separated in multiple parts, and we will cover each of them below.

5.1. Part 1

In this part we connect our client to our server using Fhir.Rest.

```
// Part 1

      // Create a context usign FHIR R4
      FhirContext ctx = FhirContext.forR4();

      // create an header containing the api key for the httpClient
      Header header = new BasicHeader("x-api-key", "api-key");
      ArrayList<Header> headers = new ArrayList<Header>();
      headers.add(header);

      // create an httpClient builder and add the header to it
      HttpClientBuilder builder = HttpClientBuilder.create();
      builder.setDefaultHeaders(headers);

      // create an httpClient using the builder
      CloseableHttpClient httpClient = builder.build();

      // Set the httpClient to the context using the factory
      ctx.getRestfulClientFactory().setHttpClient(httpClient);

      // Create a client
      IGenericClient client = ctx.newRestfulGenericClient("url");
```

In order to connect to your server you need to change the line :

```
Header header = new BasicHeader("x-api-key", "api-key");
```

And this line :

```
IGenericClient client = ctx.newRestfulGenericClient("url");
```

The 'url' is an endpoint while the "api-key" is the api key to access your server.

Note that if **you are not using** an InterSystems server you may want to check how to authorize your acces if needed.

Just like that, we have a FHIR client capable of direct exchange with our server.

5.2. Part 2

In this part we create a Patient using Fhir.Model and we fill it with a HumanName, following the FHIR convention, use and family are string and given is a list of string. The same way, a Patient can have multiple HumanNames so we have to put our HumanName in a list before puting it into our newly created Patient.

```
// Part 2

      // Create a patient and add a name to it
      Patient patient = new Patient();
      patient.addName()
         .setFamily("FamilyName")
         .addGiven("GivenName1")
         .addGiven("GivenName2");

      // See also patient.setGender or setBirthDateElement

      // Create the resource patient on the server
        MethodOutcome outcome = client.create()
         .resource(patient)
         .execute();

      // Log the ID that the server assigned
      IIdType id = outcome.getId();
      System.out.println("");
      System.out.println("Created patient, got ID: " + id);
      System.out.println("");
```

After that, we need to save our new Patient in our server using our client.

Note that if you start Client.java multiple times, multiple Patients having the name we choosed will be created. This is because, following the FHIR convention you can have multiple Patient with the same name, only the id is unique on the server.

Check the doc for more information.

Therefore we advise to comment the line after the first launch.

5.3. Part 3

In this part we get a client searching for a Patient named after the one we created earlier.

```
// Part 3

      // Search for a single patient with the exact family name "FamilyName" and the
exact given name "GivenName1"
      patient = (Patient) client.search()
         .forResource(Patient.class)
         .where(Patient.FAMILY.matchesExactly().value("FamilyName"))
         .and(Patient.GIVEN.matchesExactly().value("GivenName1"))
         .returnBundle(Bundle.class)
         .execute()
         .getEntryFirstRep()
         .getResource();

      // Create a telecom for patient
      patient.addTelecom()
```

```
         .setSystem(ContactPointSystem.PHONE)
         .setUse(ContactPointUse.HOME)
         .setValue("555-555-5555");

      // Change the patient given name to another
      patient.getName().get(0).getGiven().set(0,  new StringType("AnotherGivenName"))
;

      // Update the resource patient on the server
      MethodOutcome outcome2 = client.update()
         .resource(patient)
         .execute();
```

Once we found him, we add a phone number to his profile and we change his given name to another.

Now we can use the update function of our client to update our patient in the server.

5.4. Part 4

In this part we want to create an observation for our Patient from earlier, to do this we need his id, which is his unique identifier.

From here we fill our observation and add as the subject, the id of our Patient.

```
// Part 4

      // Create a CodeableConcept and fill it
      CodeableConcept codeableConcept = new CodeableConcept();
      codeableConcept.addCoding()
         .setSystem("http://snomed.info/sct")
         .setCode("1234")
         .setDisplay("CodeableConceptDisplay");

      // Create a Quantity and fill it
      Quantity quantity = new Quantity();
      quantity.setValue(1.0);
      quantity.setUnit("kg");

      // Create a Category and fill it
      CodeableConcept category = new CodeableConcept();
      category.addCoding()
         .setSystem("http://snomed.info/sct")
         .setCode("1234")
         .setDisplay("CategoryDisplay");

      // Create a list of CodeableConcepts and put category into it
      ArrayList<CodeableConcept> codeableConcepts = new ArrayList<CodeableConcept>();
      codeableConcepts.add(category);

      // Create an Observation
      Observation observation = new Observation();
      observation.setStatus(Observation.ObservationStatus.FINAL);
      observation.setCode(codeableConcept);
      observation.setSubject(new Reference().setReference("Patient/" + ((IIdType) out
come2.getId()).getIdPart()));
      observation.setCategory(codeableConcepts);
```

```
      observation.setValue(quantity);

      System.out.println("");
      System.out.println("Created observation, reference : " + observation.getSubject
().getReference());
      System.out.println("");

       // Create the resource observation on the server
      MethodOutcome outcome3 = client.create()
         .resource(observation)
         .execute();

      // Print the response of the server
      System.out.println("");
      System.out.println("Created observation, got ID: " + outcome3.getId());
      System.out.println("");
```

Then, we register using the create function our observation.

5.5. Conclusion of the walkthrough

If you have followed this walkthrough you now know exactly what Client.java does, you can start it and check in your server your newly created Patient and Observation.

To start it, open a VSCode terminal and enter :

```
dotnet run
```

You should see some information on the Patient we created and his observation.

If you are using an Intersystems server, go to API Deployement, authorize yourself with the api key and from here you can GET by id the patient and the observation we just created.

6. How to start coding

This repository is ready to code in VSCode with InterSystems plugins.
Open Client.java to start coding or using the autocompletion.

7. What's inside the repo

7.1. Dockerfile

A dockerfile to create a dot net env for you to work on.

Use docker build . to build and reopen your file in the container to work inside of it.

7.2. .vscode/settings.json

Settings file.

7.3. .vscode/launch.json

Config file if you want to debug

#FHIR #Java #REST API #InterSystems IRIS #Open Exchange #VSCode
Check the related application on InterSystems Open Exchange