Interoperability Code Tutorial

TABLE OF CONTENTS

C#

Setting up the Development Environment

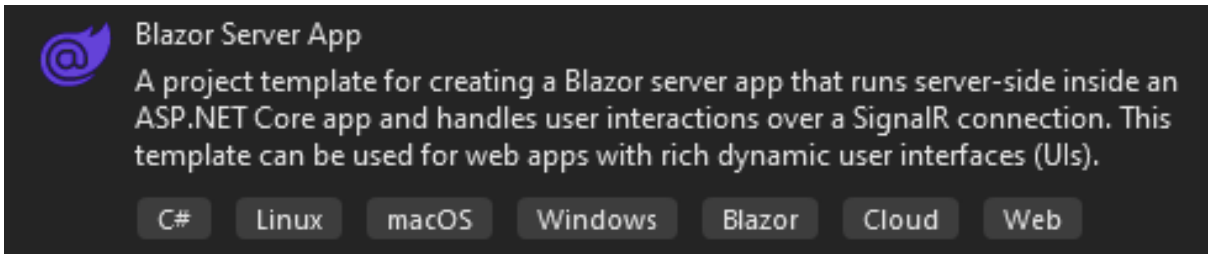
It is recommended to use Visual Studio 2022 (any edition) from Microsoft to develop your application.

The following packages from the Visual Studio Installer are required to develop a web-app:

- .NET desktop development

- ASP.NET and web development

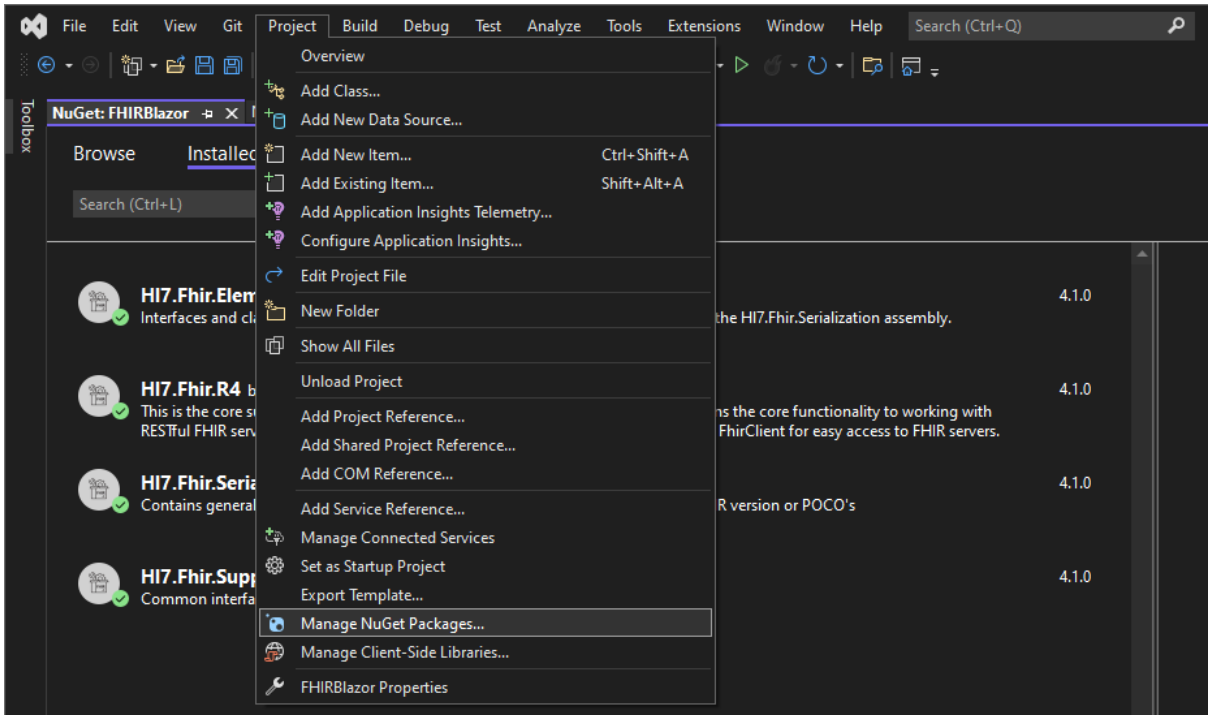
The tutorial application will be based on the Blazor Server App project template which is selectable when creating a new project.



The tutorials will not go into the specifics of Blazor and are mainly concerned with the specifics of implementing FHIR in C#.

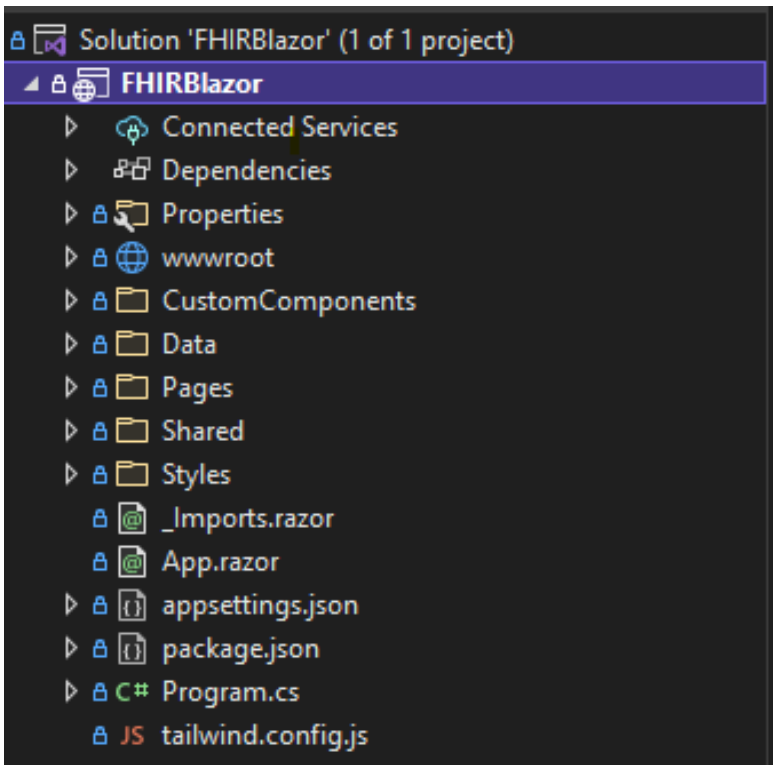
The following NuGet packages are required and heavily used in these tutorials:

- **Hl7.Fhir.ElementModel** by Firely
- **Hl7.Fhir.R4** by Firely
- **Hl7.Fhir.Serialization** by Firely
- **Hl7.Fhir.Support** by Firely



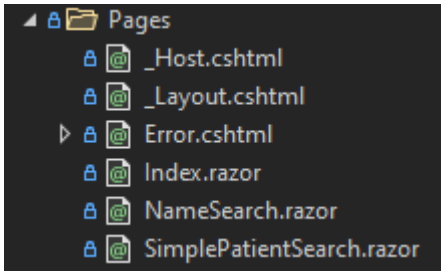
Project Directory

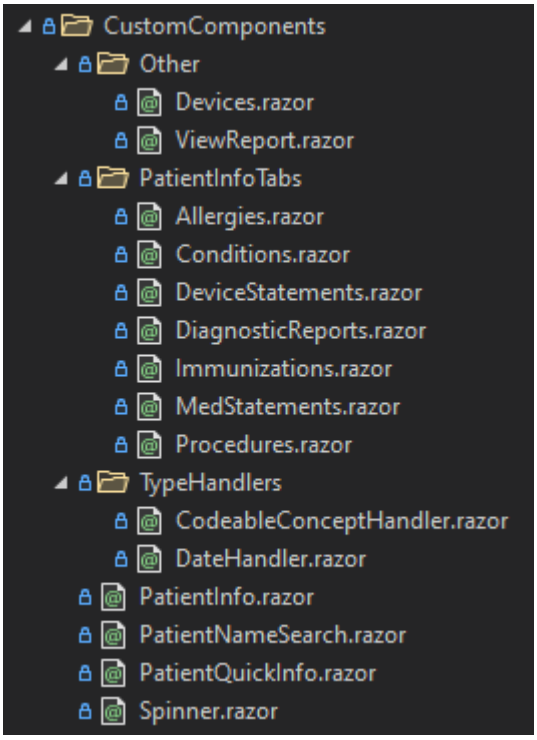
The image below shows an outline of the solution directory.



The most important folders to note are:

1. CustomComponents: contains all the components that are used to create the various pages.
2. Pages: contains basic pages of the site.





You will find FHIR and IPS related code under the CustomComponents folder. Most references to code will be pointing to this directory. The below table is quick reference guide should you get lost.

File	Description
CustomComponents/PatientInfoTabs/Allergies.razor	Code to retrieve a patients allergies
CustomComponents/PatientInfoTabs/Conditions.razor	Code to retrieve a patients conditions. ***This file has indepth comments, and the concepts covered here can be applied to all other patient info tabs***
CustomComponents/PatientInfoTabs/DeviceStatements.razor	Code to retrieve devices a patient has used
CustomComponents/PatientInfoTabs/DiagnosticReports.razor	Code to retrieve a patients diagnostic reports
CustomComponents/PatientInfoTabs/Immunizations.razor	Code to retrieve a

	patients immunizations
CustomComponents/PatientInfoTabs/MedStatements.razor	Code to retrieve a patients medical statements
CustomComponents/PatientInfoTabs/Procedures.razor	Code to retrieve all procedures perfomred on a patient
CustomComponents/TypeHandlers/CodeableConceptHandler.razor	Component to handle the FHIR CodeableConcept
CustomComponents/TypeHandlers/DateHandler.razor	Component to handle the various ways Dates can be stored in DataType.
CustomComponents/PatientInfo.razor	When a patient is selected from the search page this page will show all the info. Each piece of information is contained in its own razor page and then added to this page
CustomComponents/PatientNameSearch.razor	The main page where you can input a patient name and query results
CustomComponents/PatientQuickInfo.razor	When you click on a patient name in the search page you will get a small box with basic info. This is produced by this razor page.
CustomComponents/Spinner.razor	Used to indicate when a page is loading.
CustomComponents/Other/ViewReport.razor	When a report in DiagnosticReports.razor is clicked more information will be shown. This page is responsible for that.

CustomComponents/Other/Devices.razor	Code to retrieve devices attatched to a patient

Connecting to a Server

For the purposes of this tutorial the HAPI FHIR Test/Demo Server R4 Endpoint will be used.

To start off we must add the following code to the top of the page which is used to import the NuGet packages so we can use the built-in functions:

```
@using Hl7.Fhir.Model;
@using Hl7.Fhir.Rest;
@using Hl7.Fhir.Serialization;
```

With the NuGet packages it is relatively easy to connect to the server and fetch results. There are two components to connecting to a FHIR Server.

1. Settings: configuring various settings for output etc.

```
var settings = new FhirClientSettings
      {
        Timeout = 0,
        PreferredFormat = ResourceFormat.Json,
        VerifyFhirVersion = true,
        PreferredReturn = Prefer.ReturnMinimal
      };
```

2. Defining the client (link to server)

```
var client = new FhirClient("http://hapi.fhir.org/baseR4/");
```

With the above 2 defined we can now fetch a patient from the server.

Fetching a Patient

There are various ways a patient can be fetched. The two we will focus on is **fetching by patient ID** and **fetching by patient name**.

Fetching by Patient ID

To search by patient ID we must create a variable that will invoke `client` and append `Read` to it as shown below.

```
//Read the current version of a Patient resource
var pat_A = client.Read<Patient>("Patient/" + patientID);
```

- `client` refers to the server,
- `Read` lets the client know we want to read data.
- `<Patient>` specifies we want to read patient data
- `("Patient/" + patientID)` is what we append to the end of the link in `client`. `patientID` must be an integer.

Please note that for the hapi.fhir.org server not all patient IDs point to a patient, some may point to a pracitioner. 1104290 is an example one you can use that points to a patient.

Once fetched, the patient information will be accessible in the variable. Knowing how to access this information is one of the most important aspects to FHIR as it is not stored as it may appear. For example, name is stored as an array so to get the patients name you would invoke `pat_A.Name[0]`

Fetching by Patient Name

Searching by patient name will generate a number of results as names are not unique like the patient ID. Instead of returning one patient the server returns a bundle of patients which can then be interated through in a table or other method of displaying data.

To fetch by name we must define the search parameters.

```
var searchParam = new SearchParams()
        .Where("name:given=" + inputPatientName)
        //.OrderBy("birthdate", SortOrder.Descending)
        //.SummaryOnly().Include("Patient:organization")
        .LimitTo(maxNumOfEntries);
```

As seen in the commented code we can use other parameters. For the sake of simplicity we will focus on only two `.Where` and `.LimitTo`

`.Where` defined what we are searching for. In our case it is the given name of the patient.

`.LimitTo` allows us to control how many results we get, This is useful for controlling the size of a table.

With the search parameters defined we can execute the search with the following code.

```
Bundle results = client.Search<Patient>(searchParam);
```

The results bunde can then be iterated through using a for loop.

More information on search parameters can be found here: https://www.hl7.org/fhir/search.html

Basic Patient Information

After searching for and selecting a patient we can see a variety of personal patient information. On the `PatientInfo.razor` page we search for the patient

We first define our variables to store the results and couple of other things.

```
[Parameter] public string PatientID { get; set; }
private Patient patient;
private Boolean loading = true;
```

Note: the test server may not be live at all times so a backup server recommended is `server.fire.ly`. You may want to discover other test servers here.

/FHIR-Blazor/CustomComponents/PatientInfo.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Storing the patient result
3. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/PatientInfo.razor

Following this we read the patient from the server:

```
//Execute the search on the server and store results in the patient variable
patient = client.Read<Patient>("Patient/" + PatientID);
//Page is done loading
loading = false;
```

/FHIR-Blazor/CustomComponents/PatientInfo.razor

We can then access basic personal information of the patient using the following:

Code	Description
patient.Name	Name of the patient
patient.Gender	Gender
patient.BirthDate	When the patient was born
patient.Telcom	patient telcom information. Is a list, so use patient.Telcom.First().Value to retrieve the number, assuming the list is populated.

Other Patient Information

Code	Description
patient.Name[0].Family	Family name
patient.Name[0].Given	Loop through this to get all the given names for a patient
patient.communication	Preferred language
patient.maritalstatus	Current marital status
patient.generalPractitioner	reference to their nominated primary care provider
patient.deceased	If the patient has passed away, if not present, assume no.

Patient Address

Code	Description
patient.Address.First().Use	Address type (home, work etc.)
patient.Address.First().Country	Country
patient.Address.First().State	State
patient.Address.First().City	City
patient.Address.First().Line.First()	Street Address
patient.Address.First().PostalCode	Post Code

Emergency Contacts

Code	Description

patient.contact	A list of emergency contacts
contact.name.first().value	Name
contact.gender	Gender
contact.Telecom	Uses the same format as patient telecom seen above
contact.address	The address of the emergency contact

AllergyIntollerance

As the name suggests, this data is about a patients known allergies and intolerances to things such as food and medication. The HL7 FHIR packages make handling this information very easy. Previously when a patient is fetched, all the data alongside

Before we can fetch a patients allergies we must initialise some variables to store our information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.AllergyIntolerance[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Allergies.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is AllergyIntolerance)
4. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Allergies.razor

Following this, we execute the search in the following order:

1. Define the search parameters
2. Execute the search

3. Collect the results

```
//Define search parameters
var q = new SearchParams()
    .Where("patient=" + PatientID)
    .LimitTo(100);

//Execute search
search = client.Search<AllergyIntolerance>(q);
results = new AllergyIntolerance[search.Entry.Count];

//Collect results
for (int i = 0; i < search.Entry.Count; i++)
      {
        results[i] = (AllergyIntolerance)search.Entry[i].Resource;
      }
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Allergies.razor

To access the results we can loop through the `results` variable and print data using the following

Code	Description
allergy.Code	Prints code of the allergy
allergy.Reaction.First()	This will likely contain multiple reactions in an array so to print you must loop through the results using `foreach()` . See the example below this table. This will print the reactions.
allergy.Category.First()	Prints the category of the allergy.
allergy.Onset	Prints the onset date of the allergy. This utilises the DateHandler to produce the result in readable form

Allergy Reactions

```
@if (allergy.Reaction.Count > 0)
                {
                  var count = allergy.Reaction.First().Manifestation.Count;
                  var temp = 1;
                  @foreach (var reaction in allergy.Reaction.First().Manifestation)
                  {
                    if (temp >= count)
                    {
                      <a>@reaction.Coding.First().Display</a>
                    }
                    else
                    {
                      <a>@reaction.Coding.First().Display, </a>
                    }
```

```
                    temp++;
                  }
                }
                else
                {
                  <a></a>
                }
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Allergies.razor

Reactions are stored in `allergy.Reaction.First().Manifestation` and must be looped as reaction is a list in the FHIR standard.

Medication Statements

Any statements relating to medication a patient is prescribed are contained here. The process for fetching the data is the same as for Allergies with a minor change to the code.

Before we can fetch a patients medication statements we must initialise some variables to store our information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.MedicationStatement[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/MedStatements.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is MedicationStatement)
4. Used to indicate when the page is loading

After setting up the `FhirClient()` like before , we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

```
//Define search parameters
var sParams = new SearchParams()
      .Where("subject=" + PatientID)
      .LimitTo(100);

//Execute the search
search = client.Search<MedicationStatement>(sParams);
results = new MedicationStatement[search.Entry.Count];

//for every search result
for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (MedicationStatement)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/MedStatements.razor

We can then loop through the results in our table using the values below.

Code	Description
medstatement.Medication	Name of the medication, also includes the amount given
medstatement.Dosage	Dosage, in terms of how much of the listed medication per day (eg. 1 tablet twice daily)
medstatement.ReasonCode	Medical reason for the medication
medstatement.Id	ID of the record

Diagnostic Reports

Any diagnostic report relating to a patientt when they make a visit to a doctor or lab is stored here . The process for fetching the data is the same as for the previous with a minor change to the code.

Before we can fetch a patients diagnostic reports we must initialise some variables to store our information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.DiagnosticReport[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/DiagnosticReports.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request

2. Stores the search results initially
3. Storing the results (note the type is DiagnosticReport)
4. Used to indicate when the page is loading

After setting up the `FhirClient()` like before , we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

```
//Define search parameters
var sParams = new SearchParams()
    .Where("patient=" + PatientID)
    .LimitTo(100);

//Execute search
search = client.Search<DiagnosticReport>(sParams);
results = new DiagnosticReport[search.Entry.Count];

//for every search result
for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (DiagnosticReport)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/DiagnosticReports.razor

We can then loop through the results in our table using the values below.

Code	Description
report.Code	Name of the report
report.Category	Category the report fits under (eg. Labratory)
report.Id	ID of the record

We have added functionality in this table to click into and view the specific details of the report. This will open a dialogue box and fetch the report data in a simular manor to before.

```
//Define search parameters
[Parameter] public string ReportID { get; set; }
private DiagnosticReport report;
```

```
private Observation[]? observations;
private bool loading = true;

//Setup the FHIR client
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);

//Search and collect the results
public async ValueTask<Observation[]> GetObservationData(DiagnosticReport report)
{
  Observation[] observs = new Observation[report.Result.Count];


  this.loading = true;
  Parallel.For(0, report.Result.Count, i =>
  {
    observs[i] = client.Read<Observation>(report.Result[i].Reference);
  });

  this.loading = false;
  Console.WriteLine("Observations found");

  return observs;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/DiagnosticReports.razor

Note that we search for `<Observation>` for the report we want.

Code	Description
observation.Code	Depends on the report but can be a questions from a survey or observations
observation.Category	Category the observation fits under
observation.Issued	Issue date of the observation if applicable
observation.Id	ID of the entry

Immunizations

This will retrieve any vaccines and immunizations listed on a patients record.

First, we initialise variables to store information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.Immunization[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Immunizations.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is Immunization)
4. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Immunizations.razor

Following this, we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

```
//Define search parameters
var sParams = new SearchParams()
    Where("patient=" + PatientID)
    .LimitTo(100);

//Execute the search
search = client.Search<Immunization>(sParams);
results = new Immunization[search.Entry.Count];

//Collect the results
for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (Immunization)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Immunizations.razor

To access the results we can loop through the `results` variable and print data using the following

Code	Description
immunization.VaccineCode	Immunization identifier/name

immunization.Site	Location on the body where the immunization was given
immunization.DoseQuantity.Code	The dose given
immunization.occurance	Date the immunization was given
immunization.ExpirationDate	Date the immunization expires
immunization.Id	ID of the immunization record

Conditions

This will retrieve any conditions a patient has been diagnosed with

First, we initialise variables to store information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.Condition[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Conditions.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is Condition)
4. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Conditions.razor

Following this, we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

```
var sParams = new SearchParams()
  .Where("subject=" + PatientID)
  .LimitTo(100);

search = client.Search<Condition>(sParams);
results = new Condition[search.Entry.Count];

for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (Condition)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Conditions.razor

Code	Description
condition.Code	Name/identifier of the condition
condition.Code.Text	More details about the condition
condition.Onset	When the condition was first found in the patient
condition.Severity	Severity
condition.ClinicalStatus	Current status of the condition
condition.Category	Category the conditions sits under
condition.Id	ID of the record

History of Procedures

This will retrieve any procedures that a patient has undergone

First, we initialise variables to store information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.Procedure[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Procedures.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is Procedure)

4. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Procedures.razor

Following this, we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

```
//Define search parameters
var sParams = new SearchParams()
  .Where("patient=" + PatientID)
  .LimitTo(100);

//Execute the search
search = client.Search<Procedure>(sParams);
results = new Procedure[search.Entry.Count];

//Collect the results
for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (Procedure)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Procedures.razor

```
  search = client.Search<Condition>(sParams);
  results = new Condition[search.Entry.Count];

  for (int i = 0; i < search.Entry.Count; i++)
  {
    results[i] = (Condition)search.Entry[i].Resource;
  }
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/Procedures.razor

Code	Description
procedure.Code	Medical name of the procedure

procedure.ReasonCode	Reason for the procedure
procedure.BodySite	Where on the body the procedure was performed
procedure.Category	Category of the procedure
procedure.Performed	Date performed
procedure.Id	ID of the record

Device Use Statements

This will retrieve any medical devices that the patient has used.

First, we initialise variables to store information.

```
[Parameter] public string PatientID { get; set; }
  private Bundle search = new Bundle();
  private Hl7.Fhir.Model.DeviceUseStatement[]? results;
  private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/DeviceStatements.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is DeviceUseStatement)
4. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/DeviceStatements.razor

Following this, we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

```
//Define search parameters
var sParams = new SearchParams()
      .Where("patient=" + PatientID)
      .LimitTo(100);

//Execute the search
search = client.Search<DeviceUseStatement>(sParams);
results = new DeviceUseStatement[search.Entry.Count];

//Collect the results
for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (DeviceUseStatement)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/PatientInfoTabs/DeviceStatements.razor

Medical Devices

This will retrieve any medical devices attatched to a patient. Not currently being shown on the Patient Info page. File found in the Other folder.

First, we initialise variables to store information.

```
[Parameter] public string PatientID { get; set; }
private Bundle search = new Bundle();
private Hl7.Fhir.Model.Device[]? results;
private Boolean loading = true;
```

/FHIR-Blazor/CustomComponents/Other/Devices.razor

Starting from the top, these variables are used for:

1. Storing the patient ID which is used to make the request
2. Stores the search results initially
3. Storing the results (note the type is Device)
4. Used to indicate when the page is loading

We then setup the `FhirClient()` with our chosen server.

```
private static string serverURL = "http://hapi.fhir.org/baseR4";
FhirClient client = new FhirClient(serverURL);
```

/FHIR-Blazor/CustomComponents/Other/Devices.razor

Following this, we execute the search in the following order:

1. Define the search parameters
2. Execute the search
3. Collect the results

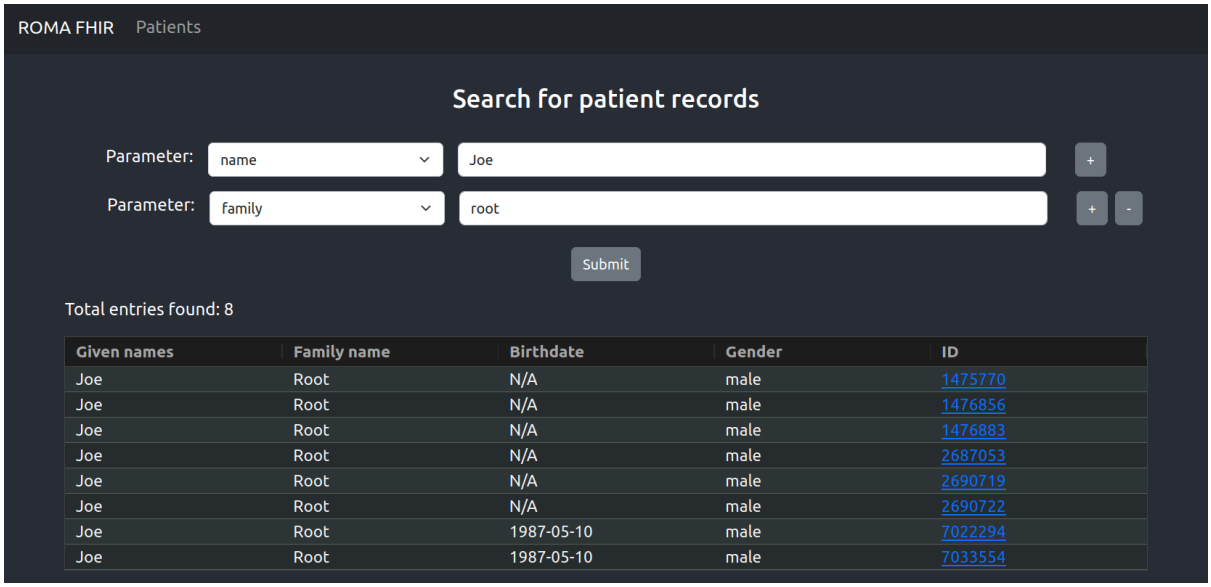
```
//Define search parameters
var sParams = new SearchParams()
  .Where("patient=" + PatientID)
  .LimitTo(100);

//Execute the search
search = client.Search<Device>(sParams);
results = new Device[search.Entry.Count];

//Collect the results
for (int i = 0; i < search.Entry.Count; i++)
{
  results[i] = (Device)search.Entry[i].Resource;
}
```

/FHIR-Blazor/CustomComponents/Other/Devices.razor

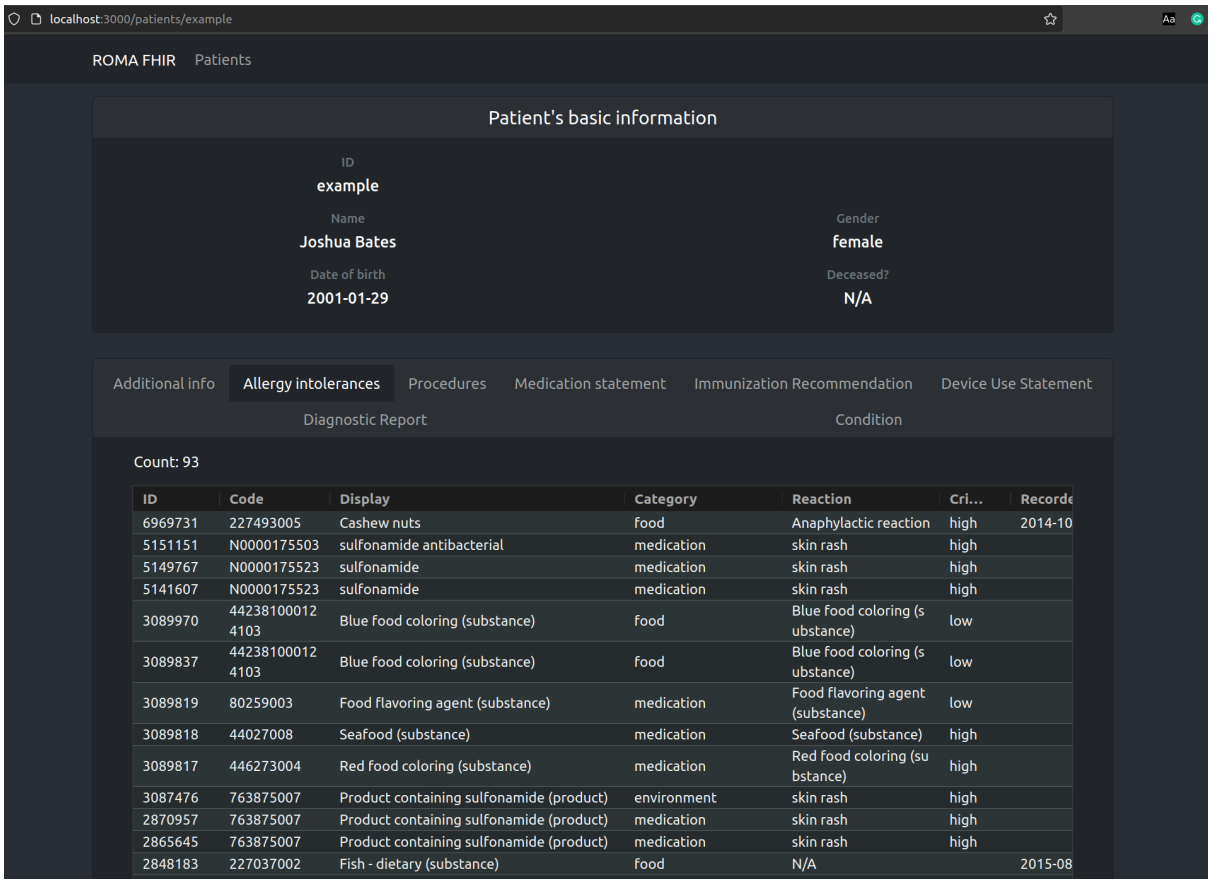
Code	Description
device.DeviceName	Name of the device (have to loop through this variable)
device.Manufacturer	Device manufacturer
device.ManufacturerDate	Date the device was produced
device.ExpirationDate	Date the device expires
device.ID	ID of the record

JavaScript

What it looks like:



/patients - Query patients with a set of parameters

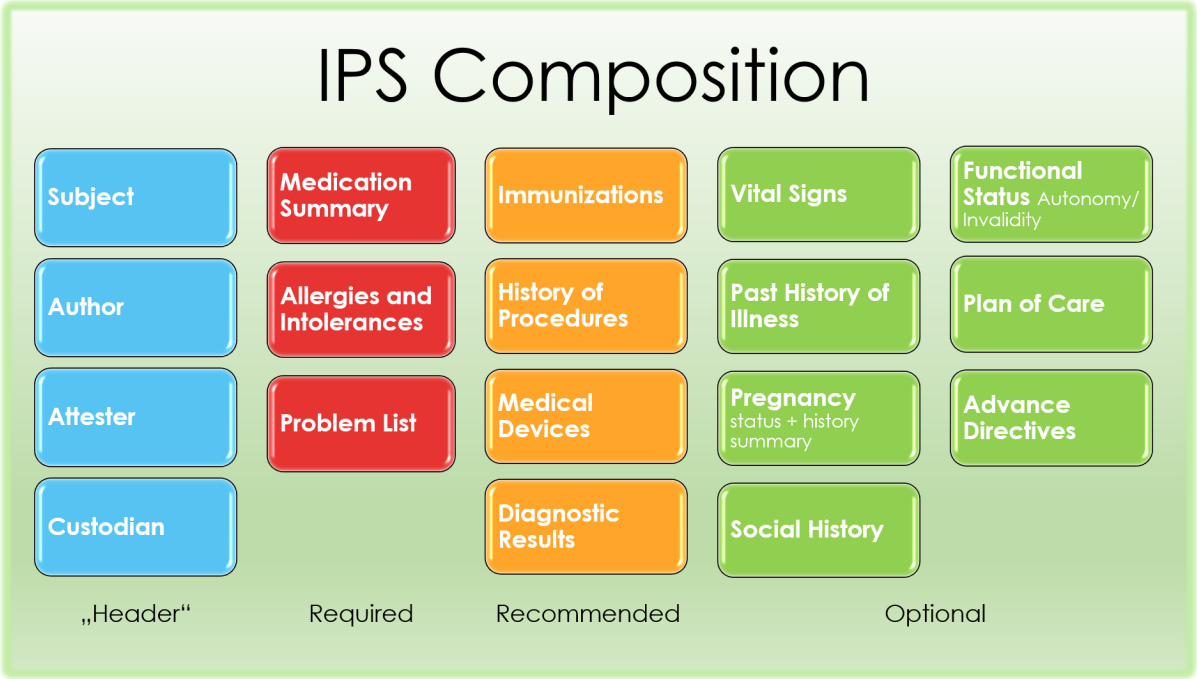


/patients/:id - Details of a patient

Introduction

This React-based application consists of demonstrations of basic interaction with the FHIR test server. Through this app, users can *query* patients and *view* essential elements (from Header to Recommended sections described in the image below) of an International Patient Summary **(IPS).**

This tutorial will discuss and focus on how all of that happens behind the scene.



https://hl7.org/fhir/uv/ips/

Setting up the Development Environment

Prerequisite

- Package manager: To install the packages for this app, a package manager such as npm is required. As that being said, npm is used throughout this tutorial.
- IDE: Use the one that you are most comfortable with. Here, we used Visual Studio Code.
- Source code: To begin the tutorial, the source code should be already available on your machine. Otherwise, download it from here.

Setting up

1. Firstly, make sure you are in the **roma/FHIR-React** directory:

   `cd path/to/the/project/roma/FHIR-React`

2. Install the dependencies/packages. `--force` flag may be required if the system is too strict about the packages being installed:

   `npm install`

3. Start the application:

   `npm start`

4. The app may automatically pop up in your default browser or you can see it at http://localhost:3000 by default settings.

Utilised Packages

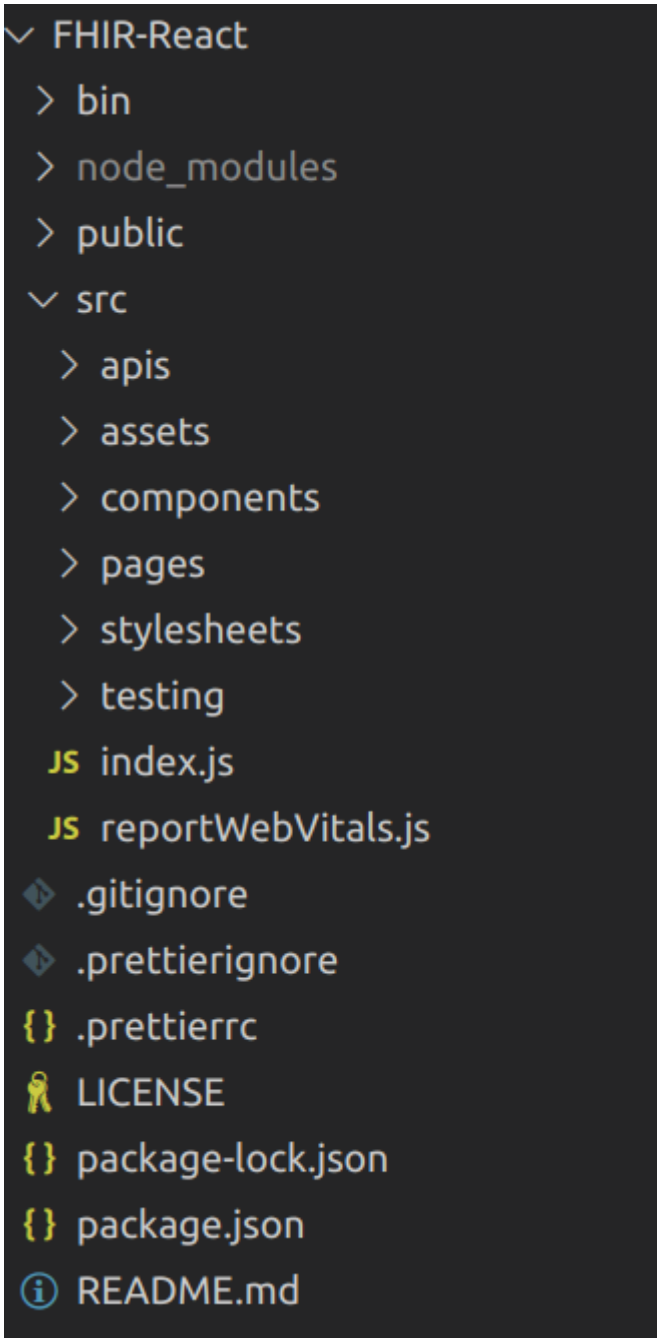
- React: React is a free and open-source front-end JavaScript library for building user interfaces based on UI components. It is maintained by Meta and a community of individual developers and companies
- Ag-grid-react: provides interactive and modern-look tables to display data. Documentation: https://ag-grid.com/react-data-grid/getting-started/
- Axios: is a promised-based HTTP client for JavaScript. It's able to make HTTP requests from the browser and handle the transformation of request and response data
- Reactstrap: contains React Bootstrap components that favour composition and control
- Bootstrap: provides styling

Directory structure

Some crucial files/folders are:

`package.json`	contains the required dependencies for this app which `npm` will rely on to install.
`/src/apis`	functions to interact with each specific API endpoint or the server in general. And based on the endpoint, they are separated into different files.
`/src/asset`	images, logos, videos, etc.
`/src/components`	contains components that can be added to a page and may be reused

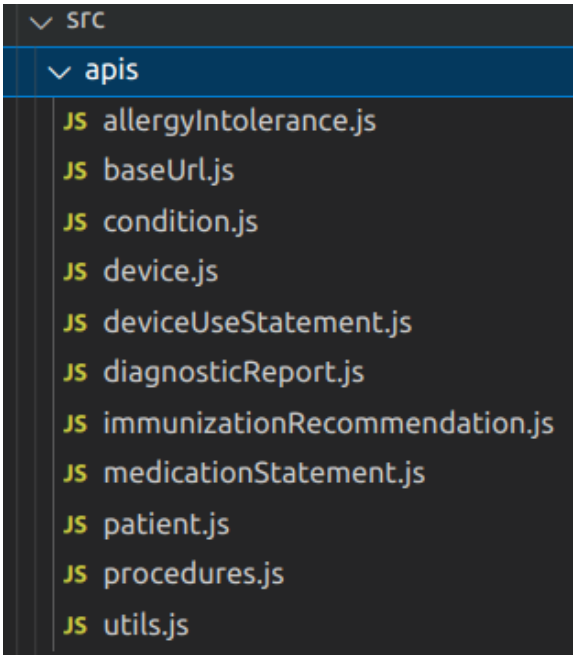
`/src/pages`	define the content of each URL path
`/src/stylesheets`	CSS files
`/src/testings`	test code that makes sure the app is running as expected in the tests. This is not utilised in the scope of this project



Utilised packages

Interacting with the Server

In this part, we will discuss in depth how the app interacts with the FHIR server such as getting various data of a patient, querying patients, etc. The `/apis` folder contains all the functions for this purpose.



Note:

Most of the files have very similar structures and overlap functionalities. For the purpose of simplicity, this tutorial will only walk through some unique files and functions.

Below are some of the most crucial parts in `/apis` that you should know. Having embraced what they do, you'll have little difficulty in understanding the rest.

baseUrl.js

For the purposes of this tutorial, the HAPI FHIR Test Server R4 Endpoint will be used. The URL of the test server is specified in `baseUrl.js` as `BASE_URL` which will be imported by other files. This ensures keeping the app uniform, meaning that every interaction of the app must be with the same server.

```
// const BASE_URL = 'https://server.fire.ly/'; // Backup test server
const BASE_URL = 'http://hapi.fhir.org/baseR4/'
export default BASE_URL;
```

Note: the test server may not be live at all times so a backup server recommended is `server.fire.ly`. You may want to discover other test servers here.

patient.js

searchPatient() queries all the patients that match the search values. It has 2 main steps:

- Constructing a complete URL by appending given arrays of parameter keys and values. For example, the final URL string with 2 parameters, gender and name, should look like this: `https://hapi.fhir.org/baseR4/Patient?gender=male&name=bob`
- Request and return response data

getPatient() gets a patient's basic data.

- Add the patient's ID to the URL with patient endpoint. The final string should look like this: `https://hapi.fhir.org/baseR4/Patient/123`
- Request and return response data

device.js

getDeviceNames() is a bit more special because it returns a `Promise`. It is done this way because, in src/components/DeviceUseStatement.js, `Promise.all()` is used to resolve an array of the results of those input promises from getDeviceNames() i.e. guaranteeing a "fully responded " array of device names.

```
// src/components/DeviceUseStatement.js, Line 64
Promise.all(response.entry.map(getDeviceNames))
```

Client-side: More about Components

Note: Most of the files have very similar structures and overlap functionalities. Some crucial, significant parts will be discussed below. Having read and understood these parts, you should have little difficulty understanding the rest.

App.js

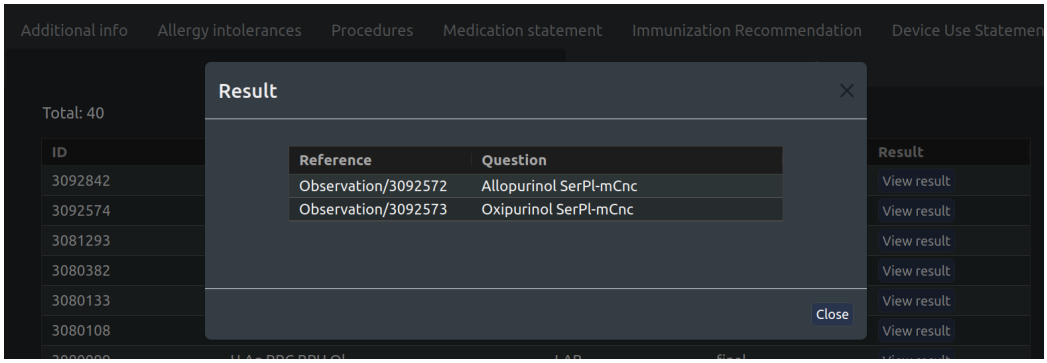
```
function App() {
  const AppConstantElements = (
    <div className="App">
      <SiteNavBar />
      <Container>
        <Outlet></Outlet>
      </Container>
    </div>
  );

  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={AppConstantElements}>
          <Route path="patients/:id" element={<PatientInfo />} />
          <Route path="patients" element={<Patients />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
```

`SiteNaveBar` is the app's Navigation Bar that will appear at the top of the page at all times. `BrowserRouter` is utilised to render components based on the URL path which will all be contained in the `Container`

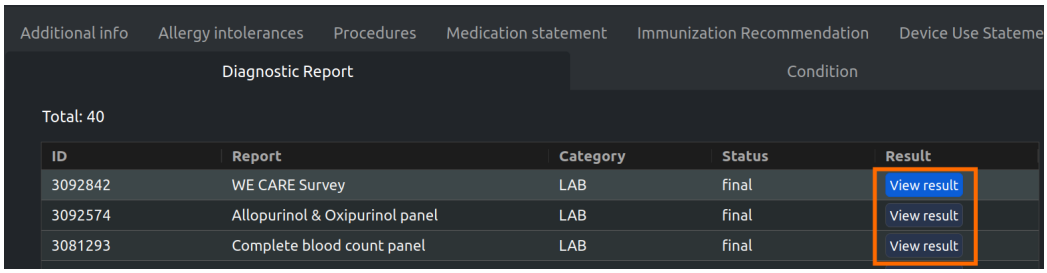
DiagnosticReport.js

- convertEntry
  - This function appears in most components
  - Used to convert multiple entries to an Object that is compatible with the AgGridReact component. Since the data from the server response is structured differently, re-mapping and error/special case handling are essential steps so that AgGridReact can read and display data properly.
- convertResult
  - Similar to convertEntry, convertResult is used to re-mapping data for the ResultModal component
- ResultModal
  - A modal to display results of selected Diagnostic Report

- It also uses 2 assistive useState hooks, `modalShow` and `clickedRowNumber`, to function.



- ResultButton
  - As can be seen in the Diagnostic Report tab, each row of the table has a View Result button in the right-most column which is the ResultButton component. Every time it is clicked, an associated result table in ResultModal will appear.




```
columnDefs: [
  { headerName: 'ID', field: 'id', width: 110 },
  { headerName: 'Report', field: 'report', minWidth: 120 },
  { headerName: 'Category', field: 'category', width: 100 },
  { headerName: 'Status', field: 'status', width: 100 },
  { headerName: 'Result', field: 'result', width: 100,
    cellRenderer: (params) =>  resultButton(params, setModalShow, setClickedRowNumber)},
],
```

  - To do this, we utilise `cellRenderer` (in `columnDefs` of AgGridReact) to create more complex HTML inside a cell. Read more here: https://www.ag-grid.com/javascript-data-grid/component-cell-renderer/

```
columnDefs: [
  { headerName: 'ID', field: 'id', width: 110 },
  { headerName: 'Report', field: 'report', minWidth: 120 },
  { headerName: 'Category', field: 'category', width: 100 },
  { headerName: 'Status', field: 'status', width: 100 },
  { headerName: 'Result', field: 'result', width: 100,
    cellRenderer: (params) =>  resultButton(params, setModalShow, setClickedRowNumber)},
],
```

DeviceUseStatement.js

This component is useful in demonstrating how to get data from more than 2 endpoints and display it properly to the client side.

The first primary data comes from /DeviceUseStatement (see `api/deviceUseStatement.js`) and the second data contains the associated device name comes from /Device (see `api/device.js`)

- Promise.all()
  - Returns a fully resolved array of device names which is later used in convertEntry()
  - As mentioned in device.js, `Promise.all()` is used to resolve an array of the results of those input promises from getDeviceNames() i.e. guaranteeing a “fully responded ” array of device names.
- convertEntry()
  - The unique thing about convertEntry() of DeviceUseStatement.js is that it receives a 2nd parameter `deviceNames`

```
const convertEntry = (entries, deviceNames) => {
  const rowData = entries.map( (entry, index) => {
    const resource = entry.resource;
    return {
      id: resource.device? resource.device.reference.split('/')[1] : 'N/A',
      name: deviceNames[index],
      derivedFrom: resource.derivedFrom? resource.derivedFrom[0].reference : 'N/A',
      source: resource.source? resource.source.reference : 'N/A',
    }
  });
  return rowData;
};
```

  - As can be seen from the code block above, `index` argument from `callbackFn` is utilised to easily access and attach an associated name from `deviceNames` parameter for each row (or each device use statement).
  - Read more about `callbackFn` here