Firely .NET SDK

Firely

Feb 23, 2021

FIRELY PRODUCTS

It's easy to start using the Firely .NET SDK in your solution: start with the right NuGet package.

CHAPTER

ONE

INSTALL VIA .NUSPEC

dotnet add package Hl7.Fhir.R4

CHAPTER

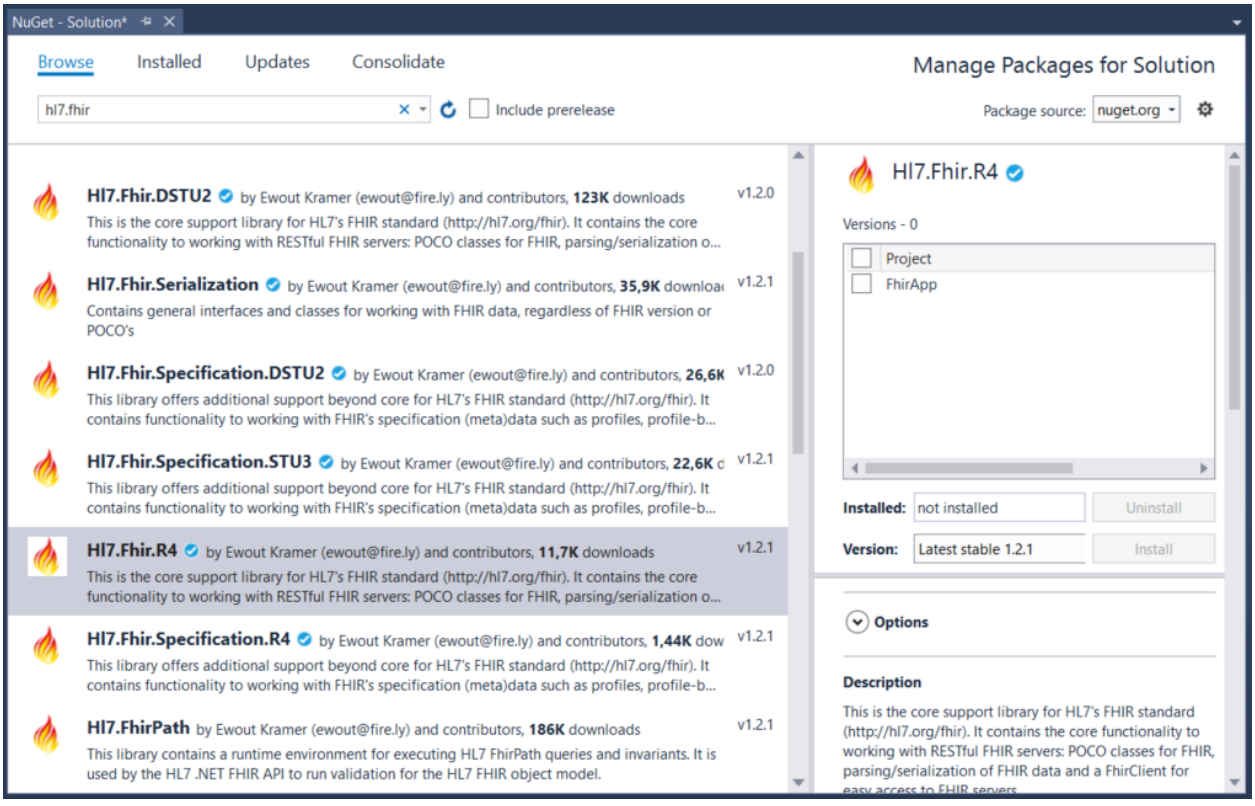
TWO

INSTALL VIA VISUAL STUDIO

Open your project, or start a new one and go to the NuGet Package Manager. The current official version of the FHIR specification is R4, but you might be developing for one of the previous versions. For each version of the specification, there is a corresponding library.

For R4, you will need the Hl7.Fhir.R4 package:

1. Choose Tools NuGet Package Manager Manage NuGet Packages for Solution...
2. Click on Browse, and type fhir into the search field.
3. Click on the package you need, in this case Hl7.Fhir.R4.



4. Check the box next to the name of your project and click on Install.

The *next section* explains how the SDK works with the FHIR model.

CHAPTER

THREE

WORKING WITH THE MODEL

`Hl7.Fhir.Model` contains model classes that correspond to the FHIR Resources and data types, like Patient and HumanName. The code is generated by the FHIR publication tool, which is used to build a new version of the FHIR website.

Note that the publication tool can only help generate the base resource classes, it can't generate classes from FHIR profiles.

In this chapter, we will explain how to work with the model, and give you some code examples. We also include a complete code example that sets up an instance of the Patient resource, filling in values for several of the fields. We conclude the chapter with a discussion of the Bundle resource type, plus some code examples of how to work with Bundles.

Add this `using` directive to your code:

```
using Hl7.Fhir.Model;
```

3.1 Model classes

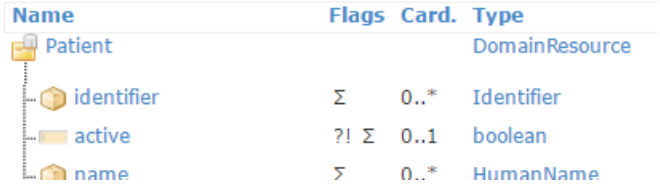
For each Resource type and data type in FHIR, the SDK contains a class in the form of a public class. Creating a new Patient resource instance, and an instance of the Identifier data type:

```
var pat = new Patient();
var id = new Identifier();
```

Important: When you are creating an instance of a resource or data type, lookup the definition in the FHIR specification to see which elements are mandatory for that particular type.

3.1.1 Class fields

The SDK classes have a field for each of the elements in the Resource or data type model. For example, the Patient resource has an `active` element:



The Patient class in the SDK has a field called `Active` that corresponds with this element:

```
namespace Hl7.Fhir.Model
{
    /// <summary>
    /// Information about an individual or animal receiving health care services
    /// </summary>
    [FhirType("Patient", IsResource=true)]
    public partial class Patient : Hl7.Fhir.Model.DomainResource, System.ComponentModel.INotifyPropertyChanged
    {
        ...
        /// <summary>
        /// Whether this patient's record is in active use
        /// </summary>
        /// <remarks>This uses the native .NET datatype, rather than the FHIR equivalent</remarks>
        [NotMapped]
        [IgnoreDataMemberAttribute]
        6 references
        public bool? Active
        {
          ...
        }
```

Likewise, the Identifier data type has an element called `use`:

Name	Flags	Card.	Type
Identifier	Σ		Element
use	?! Σ	0..1	code

And the Identifier class in the SDK has a field called `Use` that corresponds with this element:

```
namespace Hl7.Fhir.Model
{
    /// <summary>
    /// An identifier intended for computation
    /// </summary>
    [FhirType("Identifier")]
    public partial class Identifier : Hl7.Fhir.Model.Element, System.ComponentModel.INotifyPropertyChanged
    {
        ...
        /// <summary>
        /// usual | official | temp | secondary (If known)
        /// </summary>
        /// <remarks>This uses the native .NET datatype, rather than the FHIR equivalent</remarks>
        [NotMapped]
        [IgnoreDataMemberAttribute]
        0 references
        public Hl7.Fhir.Model.Identifier.IdentifierUse? Use
        {
          ...
        }
```

As you can see, the classes and fields all have inline documentation describing them.

3.2 Primitive data types

In FHIR, the data types are divided into 'primitive' and 'complex' data types. The primitive data types are types like `string`, `integer`, `boolean`, etc. that can take a single value. The complex types consist of multiple values grouped together.

Important: Primitives are not really primitive in FHIR!

Because you can extend resources **and** data types in FHIR, the SDK has provided FHIR data types for the primitive types. Where the name of the FHIR data type would conflict with existing .Net data types, the word 'Fhir' is added to the type, e.g. `FhirString`.

For each of the fields that take a primitive data type, the SDK provides you with two fields in the class. One of the fields has the same name as the element it corresponds with in the FHIR resource, e.g. `Active` in the `Patient` class. This field is of the standard .Net data type.

You can fill this field just the way you would expect:

```
var pat = new Patient();
pat.Active = true;
```

The other field has got the name of the element, with 'Element' added to it, for example `ActiveElement` in the `Patient` class. You fill this field with the FHIR data type that is in the SDK:

```
pat.ActiveElement = new FhirBoolean(true);
```

Note: Both of the statements set the same private data member of the class.

3.3 Complex data types

Complex data types in FHIR are data types that group certain values together, such as `Address`, `Identifier` and `Quantity`. The FHIR specification describes which elements are part of these data types.

The SDK has created classes for each of the data types, with fields for each of the elements. Most of the elements will be of a primitive data type, but you can also encounter complex types within a complex data type.

Filling in the fields for the primitive types is explained *in the previous paragraph*. However, if you need to fill in a field that is of a complex data type, you will need to create an instance of that type first.

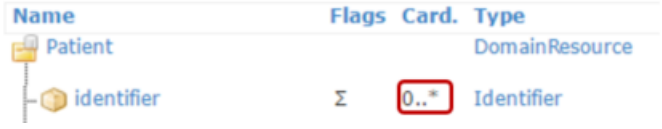
For example, if we want to fill in the data for a field of type `Identifier`, we can use this code:

```
var id = new Identifier();

id.System = "http://hl7.org/fhir/sid/us-ssn";
id.Value = "000-12-3456";
```

3.4 Lists

For elements with a maximum cardinality > 1, the SDK has a list of the type for that element.



```
public partial class Patient : Hl7.Fhir.Model.DomainResource, System.ComponentModel.INotifyPropertyChanged
{
    ...
    /// <summary>
    /// An identifier for this patient
    /// </summary>
    [FhirElement("identifier", InSummary=true, Order=90)]
    [Cardinality(Min=0,Max=-1)]
    [DataMember]
    22 references
    public List<Hl7.Fhir.Model.Identifier> Identifier
    {
      ...
    }
```

To work with data in a list, you can use the standard C# List methods.

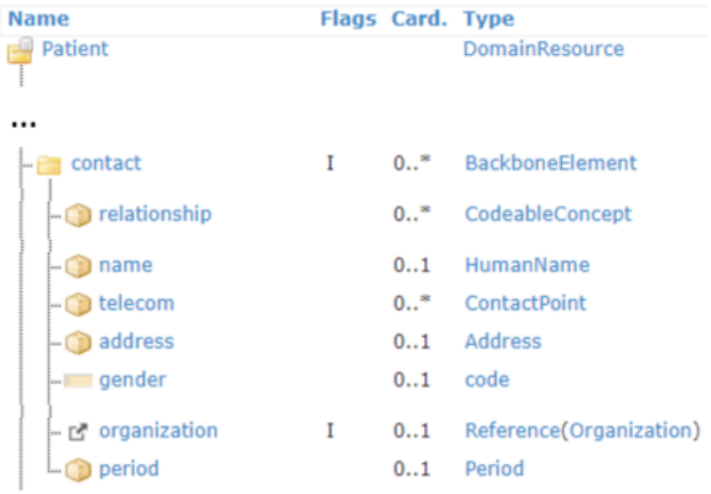
So for example, if we want to add the `Identifier` we created in the previous paragraph to the `Identifier` field of the instance of `Patient` we created earlier, we can do this:

```
pat.Identifier.Add(id);
```

Note: If you did not initialize a field before adding to the list, the SDK will create the List for you, and will not generate a NullReferenceException.

3.5 Components

Resources can have elements with a subgroup of elements in them. These are called 'BackboneElements' or 'components'. For example, the Patient resource type has a component called `contact`.



In the SDK, a component block is represented by a class within the resource type class. This subclass has the name of the field, followed by 'Component', for example `ContactComponent` in the `Patient` class:

```
namespace Hl7.Fhir.Model
{
  public partial class Patient : Hl7.Fhir.Model.DomainResource, System.ComponentModel.INotifyPropertyChanged
  {
    ...
    [FhirType("ContactComponent")]
    [DataContract]
    9 references
    public partial class ContactComponent : Hl7.Fhir.Model.BackboneElement, System.ComponentModel.INotifyPropertyChanged
    {
      ...
    }
```

Code example, adding contact details to our Patient:

```
var contact = new Patient.ContactComponent();
contact.Name = new HumanName();
contact.Name.Family = "Parks";
// setup other contact details

pat.Contact.Add(contact);
```

3.6 Enumerations

For coded types in FHIR, the elements are bound to a ValueSet. When the specification states that the ValueSet is 'Required', this means it is a fixed list of codes. The SDK provides an enumeration for each fixed ValueSet. You can use these enumerations to fill in the correct value.

The Patient resource has a fixed ValueSet for the `gender` element.

Name	Flags	Card.	Type	Description & Constraints
Patient			DomainResource	Information about an individual or animal receiving health care services Elements defined in Ancestors: id, meta, implicitRules, language, text, cc
...				
gender	Σ	0..1	code	male | female | other | unknown AdministrativeGender (Required)

Enumeration in the SDK:

```
/// <summary>
/// The gender of a person used for administrative purposes.
/// (url: http://hl7.org/fhir/ValueSet/administrative-gender)
/// </summary>
[FhirEnumeration("AdministrativeGender")]
56 references
public enum AdministrativeGender
{
    [EnumLiteral("male"), Description("Male")]
    Male,

    ...
}
```

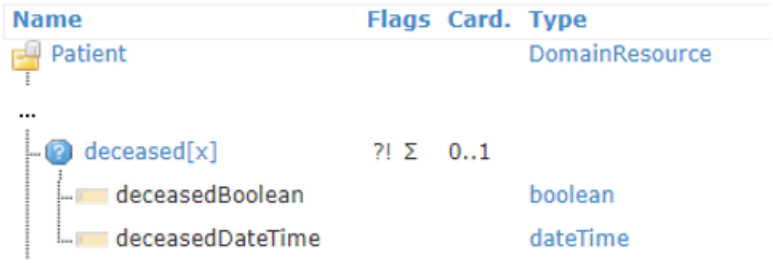
Code example, adding a gender to our Patient:

```
pat.Gender = AdministrativeGender.Male;
```

3.7 Choice properties

In the FHIR specification, you will encounter 'choice properties' for some of the resource's elements. This means that you can choose the type you fill in for that element, from the possible types listed.

For the Patient resource type for example, we have a choice for the `deceased` element:



In the SDK, you will see that the corresponding field is of type `Element`, which is the base for all data types.

```
/// <summary>
/// Indicates if the individual is deceased or not
/// </summary>
[FhirElement("deceased", InSummary=true, Order=150, Choice=ChoiceType.DatatypeChoice)]
[CLSCompliant(false)]
[AllowedTypes(typeof(Hl7.Fhir.Model.FhirBoolean),typeof(Hl7.Fhir.Model.FhirDateTime))]
[DataMember]
12 references
public Hl7.Fhir.Model.Element Deceased
{
  ...
}
```

This means that in your code, you will first have to create an instance of the data type of your choice, before you can fill in the field. For example, if we choose to use a date for the `Deceased` field of our Patient, we could implement that like this:

```
var deceased_date = new FhirDateTime("2015-04-23");
pat.Deceased = deceased_date;
```

Or, if we choose to fill in a boolean value:

```
pat.Deceased = new FhirBoolean(true);
```

The list of all available types is available here and here.

3.8 Special initializers

As you can see from the example in the previous paragraph with the `FhirDateTime` or `FhirBoolean`, for several data types, the SDK provides you with extra initialization methods. Visual Studio's IntelliSense will help you to view the possibilities while you type, or you can take a look at `Hl7.Fhir.Model` with the Object Browser to view the methods, plus their attributes as well.

For the `HumanName` data type, the SDK has added some methods to make it easier to construct a name in one go, using fluent notation:

```
pat.Name.Add(new HumanName().WithGiven("Christopher").WithGiven("C.H.").AndFamily(
↪"Parks"));
```

If you need to fill in more than the `Given` and `Family` fields, you could first construct a `HumanName` instance in this manner, and add to the fields later on. Or you could choose not to use this notation, but instead fill in all the fields the way it was explained in the other paragraphs.

3.9 Extensions

In the *Primitive data types* paragraph, we have mentioned that both resource *and* data types can be extended in FHIR. To add an extension to your data, you will have to fill in both a URL identifying the extension and a value that is valid according to the definition of the extension.

The following code example adds a place and time of birth to our Patient instance defined by standard extensions in the FHIR specification. The URL and the type of value you can fill in are listed in the definition of the extension. The definition will also tell you on what field the extension can be added.

```
var birthplace = new Extension();
birthplace.Url = "http://hl7.org/fhir/StructureDefinition/birthPlace";
birthplace.Value = new Address() { City = "Seattle" };
pat.Extension.Add(birthplace);

var birthtime = new Extension("http://hl7.org/fhir/StructureDefinition/patient-
↪birthTime",
                              new FhirDateTime(1983,4,23,7,44));
pat.BirthDateElement.Extension.Add(birthtime);
```

The extension for birthPlace is pretty straightforward to add. The URL is taken from the extension definition. The value is of type `Address`, and the extension can be added to the top-level of the Patient instance. The birthTime extension is a little more complex. This extension takes a `dateTime` value, and has to be added to the `BirthDate` field. For this field the SDK provides you with the easy way to fill it, by allowing you to set the value of `BirthDate` as a `string` – internally converting this to the `Date` type. This means you will have to use the `[fieldname]Element` construction to add extensions to the field.

3.10 Code example for Patient

With the code examples from the previous paragraphs, plus some additions, we have constructed a code example that sets up an instance of the Patient resource, with some information covering all of the topics of this section. We have tried to include different ways to fill in the fields, so you can see the possibilities and choose what suits your programming style best.

```
// example Patient setup, fictional data only
var pat = new Patient();

var id = new Identifier();
id.System = "http://hl7.org/fhir/sid/us-ssn";
id.Value = "000-12-3456";
pat.Identifier.Add(id);

var name =  new HumanName().WithGiven("Christopher").WithGiven("C.H.").AndFamily(
↪"Parks");
name.Prefix = new string[] { "Mr." };
```

(continues on next page)

(continued from previous page)

```
name.Use = HumanName.NameUse.Official;

var nickname = new HumanName();
nickname.Use = HumanName.NameUse.Nickname;
nickname.GivenElement.Add(new FhirString("Chris"));

pat.Name.Add(name);
pat.Name.Add(nickname);

pat.Gender = AdministrativeGender.Male;

pat.BirthDate = "1983-04-23";

var birthplace = new Extension();
birthplace.Url = "http://hl7.org/fhir/StructureDefinition/birthPlace";
birthplace.Value = new Address() { City = "Seattle" };
pat.Extension.Add(birthplace);

var birthtime = new Extension("http://hl7.org/fhir/StructureDefinition/patient-
↪birthTime",
                              new FhirDateTime(1983,4,23,7,44));
pat.BirthDateElement.Extension.Add(birthtime);

var address = new Address()
{
        Line = new string[] { "3300 Washtenaw Avenue, Suite 227" },
        City = "Ann Arbor",
        State = "MI",
        PostalCode = "48104",
        Country = "USA"
};
pat.Address.Add(address);

var contact = new Patient.ContactComponent();
contact.Name = new HumanName();
contact.Name.Given = new string[] { "Susan" };
contact.Name.Family = "Parks";
contact.Gender = AdministrativeGender.Female;
contact.Relationship.Add(new CodeableConcept("http://hl7.org/fhir/v2/0131", "N"));
contact.Telecom.Add(new ContactPoint(ContactPoint.ContactPointSystem.Phone, null, "
↪"));
pat.Contact.Add(contact);

pat.Deceased = new FhirBoolean(false);
```

3.11 Bundles

Although Bundle is just another resource type in FHIR, and you can fill in the values for the fields in the way that has been described in this chapter, a Bundle is still a bit special. Bundles are used to communicate sets of resources. Usually you will first encounter them when you're performing a *search interaction*. In that case, the server will send you a Bundle, and you will browse through the contents to process the data. Sometimes you will need to construct a Bundle instance and fill in the details, for example when you are going to setup a FHIR document, or want to perform a transaction, or if you're implementing the server side response to a search request.

3.11.1 Looking at the content

A Bundle resource has got some fields that contain metadata about the Bundle, such as the type of Bundle, and a total if the Bundle contains a result from a *search* or *history* interaction. The resources that are put in a Bundle, are located in the `entry` element of the Bundle resource.

Since `entry` is a 0..* element **and** a *component* block, the SDK provides you with a list of `EntryComponent` in the `Bundle.Entry` field. You can loop through that list, or use any of the standard C# List methods to work with the list.

The fully qualified URL identifying the resource that is in the entry, is stored in the `FullUrl` field of the entry. The SDK doesn't know the type of resource that is in the entry, so the data type for the `Resource` field in the entry is the base type `Resource`. You will need to cast to the actual resource type if you want to have access to the fields for that type.

Tip: You can check the `Resource.ResourceType` field first, if you don't know the type of resource for an entry.

Suppose we have performed a search interaction on Patient resources, and have stored the results in a variable called `result`. We can then loop through the resources in the `Entry` list like this:

```
foreach (var e in result.Entry)
{
        // Let's write the fully qualified url for the resource to the console:
        Console.WriteLine("Full url for this resource: " + e.FullUrl);

        var pat_entry = (Patient)e.Resource;

        // Do something with this patient, for example write the family name that's␣
↪in the first
        // element of the name list to the console:
        Console.WriteLine("Patient's last name: " + pat_entry.Name[0].Family);
}
```

3.11.2 Filling a Bundle

When constructing a Bundle, you will need to look at the definition for the Bundle resource to see which elements are mandatory, just as you would do for other resource types.

Tip: If you want to create a Bundle for a batch or transaction, you can use the helper methods in the SDK to construct the Bundle correctly, described in the transactions paragraph.

Then, for each resource you want to include in the Bundle, you will add an `EntryComponent` to the `Entry` list. This can be done by creating an instance of type `EntryComponent`, which you fill with the fully qualified URL for the resource, and the resource. Or, you could use the `AddResourceEntry` method of the `Bundle` class. This second option creates cleaner code if you only need to fill in the URL and resource. However, if you need to fill in more fields for the `EntryComponent` class, the first option can be useful.

This example shows both ways:

```
var collection = new Bundle();
collection.Type = Bundle.BundleType.Collection;

var first_entry = new Bundle.EntryComponent();
```

(continues on next page)

(continued from previous page)

```
first_entry.FullUrl = res1.ResourceBase.ToString() + res1.ResourceType.ToString() +␣
↪res1.Id;
first_entry.Resource = res1;
collection.Entry.Add(first_entry);

// adding a second entry
collection.AddResourceEntry(res2, "urn:uuid:01d04293-ed74-4f93-aa0a-2f096a693fb1");
```

In this example we create a Bundle with a general collection of resources, and have set the type accordingly. The first resource we want to add, `res1`, is a resource that already has a technical id. For now, we have constructed the FullUrl with parts of the information that's in the resource instance, but we could also have used the helper methods for `ResourceIdentity` which are in the `Hl7.Fhir.Rest` namespace. See *About resource identity* for more information. The second resource we add to this collection, `res2`, is a new resource that has not been stored, and doesn't have a technical identifier assigned to it yet. We still have to fill in the `FullUrl` field, as demanded by the `bdl-7` constraint for Bundle in the specification. This is done by creating a temporary UUID, and representing that as a fully qualified URL with the `urn:uuid:` notation.

3.12 Code example for Bundle

In this paragraph, we provide a code example that sets up a Bundle that could be the response to a search request. Next to adding the resources to the `Entry` field, we have also added some extra data such as the total number of results. At the end, we have included the loop to walk through the Bundle entries.

```
// example searchset Bundle setup, fictional data only
var search_response = new Bundle();
search_response.Type = Bundle.BundleType.Searchset;

// adding some metadata
search_response.Id = "[insert temporary uuid here, or real id if you store this␣
↪Bundle]";
search_response.Meta = new Meta()
{
        VersionId = "1",
        LastUpdatedElement = Instant.Now()
};

// we assume the search has already taken place on your database, and the resulting
// resources are available to us in a list called 'dataset'
search_response.Total = dataset.Count;

// for searches, we need to fill in the 'self' link, that represents the search␣
↪request
// as understood by the server, e.g. "http://myserver.org/fhir/Patient?name=steve"
// if you are paging the response, also fill in the other relevant links, like 'next',
// 'last', etc.
search_response.SelfLink = new Uri("[search request]");

foreach (var r in dataset)
{
        var full_url = r.ResourceBase.ToString() + r.ResourceType.ToString() + r.Id;

        // instead of using AddResourceEntry, we use the search variant to also␣
↪include
        // the search mode
```

(continues on next page)

(continued from previous page)

```
        search_response.AddSearchEntry(r, full_url, Bundle.SearchEntryMode.Match);
}

// the Bundle is now ready to be sent to the requester

// walking through the entries in the Bundle:
foreach (var e in search_response.Entry)
{
        // Let's write the fully qualified url for the resource to the console:
        Console.WriteLine("Full url for this resource: " + e.FullUrl);

        // Do something with this resource, for example write human readable text of
        // the resource to the console:
        var resource = (DomainResource)e.Resource;
        Console.WriteLine("Human readable text of this resource: " + resource.Text);
}
```

CHAPTER

FOUR

WORKING WITH REST

In this section we explain the methods of the FhirClient that are in the `Hl7.Fhir.Rest` part of the SDK package.

Add this `using` directive to your code:

```
using Hl7.Fhir.Rest;
```

The first topics in this chapter cover the settings of the FhirClient and the CRUD interactions you can perform with it. We will then give some examples of history and search interactions, and explain how to perform operations and transactions. There's also a section on helper methods for resource identities, and we end the chapter with other miscellaneous methods and helpers in the `Hl7.Fhir.Rest` namespace.

4.1 Creating a FhirClient

Before we can do any of the interactions explained in the next part, we have to create a new `FhirClient` instance. This is done by passing the url of the FHIR server's endpoint as a parameter to the constructor:

```
var client = new FhirClient("http://server.fire.ly");
```

The constructor method is overloaded, to enable you to use a URI instead of a string. As second parameter to the constructor, you can specify whether the client should perform a conformance check to see if the server has got a compatible FHIR version. The default setting is `false`.

A FhirClient works with a single server. If you work with multiple servers simultanuously, you'll have to create a FhirClient for each of them. Since resources may reference other resources on a different FHIR server, you'll have to inspect any references and direct them to the right FhirClient. Of course, if you're dealing with a single server within your organization or a single cloud-based FHIR server, you don't have to worry about this. Note: the FhirClient is not thread-safe, so you will need to create one for each thread, if necessary. But don't worry: creating an instance of a FhirClient is cheap, the connection will not be opened until you start working with it.

There's a list of publically available test servers you can use.

4.1.1 FhirClient communication options

To specify some specific settings, you add a `FhirClientSettings` to the constructor:

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

var client = new FhirClient("http://server.fire.ly", settings)
```

You can also toggle these settings after the client has been initialized.

To specify the preferred format –JSON or XML– of the content to be used when communicating with the FHIR server, you can use the `PreferredFormat` attribute:

```
client.Settings.PreferredFormat = ResourceFormat.Json;
```

The FHIR client will send all requests in the specified format. Servers are asked to return responses in the same format, but may choose to ignore that request. The default setting for this field is `XML`.

When communicating the preferred format to the server, this can either be done by appending `_format=[format]` to the URL, or setting the `Accept` HTTP header. The client uses the latter by default, but if you want, you can use the `_format` parameter instead:

```
client.Settings.UseFormatParam = true;
```

If you perform a `Create`, `Update` or `Transaction` interaction, you can request the server to either send back the complete representation of the interaction, or a minimal data set, as described in the Managing Return Content section of the specification. The default setting is to ask for the complete representation. If you want to change that request, you can set the `PreferredReturn` attribute:

```
client.Settings.PrefferedReturn = Prefer.ReturnMinimal;
```

This sets the `Prefer` HTTP header in the request to `minimal`, asking the server to return no body in the response.

You can set the timeout to be used when making calls to the server with the `Timeout` attribute:

```
client.Timeout = 120000; // The timeout is set in milliseconds, with a default of␣
↪100000
```

4.2 CRUD interactions

A FhirClient named `client` has been setup in the previous topic, now let's do something with it.

4.2.1 Create a new resource

Assume we want to create a new resource instance and want to ask the server to store it for us. This is done using `Create`.

```
var pat = new Patient() { /* set up data */ };
var created_pat = client.Create(pat);
```

Tip: See *Working with the model* for examples on how to fill a resource with values.

As you'd probably expect, this interaction will throw an Exception when things go wrong, in most cases a `FhirOperationException`. This exception has an `Outcome` property that contains an OperationOutcome resource, and which you may inspect to find out more information about why the interaction failed. Most FHIR servers will return a human-readable error description in the `OperationOutcome` to help you out.

If the interaction was successful, the server will return an instance of the resource that was created, containing the id, metadata and a copy of the data you just posted to the server as it was stored. Depending on the server implementation, this could differ from what you've sent. For instance, the server could have filled in fields with default values, if the values for those fields were not set in your request.

If you've set the `PreferredReturn` property of the FhirClient to `minimal`, the server will return the technical id and version number of the newly created resource in the headers of the response and the `Create` method will return `null`. See *Message Handlers* for an example of how to retrieve the information from the returned headers.

For the conditional version of this interaction, see conditionals.

4.2.2 Reading an existing resource

To read the data for a given resource instance from a server, you'll need its technical id. You may have previously stored this after a `Create`, or you have found its address in a ResourceReference (e.g. `Observation.Subject.Reference`).

The `Read` interaction on the FhirClient has two overloads to cover both cases. Furthermore, it accepts both relative paths and absolute paths (as long as they are within the endpoint passed to the constructor of the `FhirClient`).

```
// Read the current version of a Patient resource with technical id '31'
var location_A = new Uri("http://server.fire.ly/Patient/31");
var pat_A = client.Read<Patient>(location_A);
// or
var pat_A = client.Read<Patient>("Patient/31");

// Read a specific version of a Patient resource with technical id '32' and version␣
↪id '4'
var location_B = new Uri("http://server.fire.ly/Patient/32/_history/4");
var pat_B = client.Read<Patient>(location_B);
// or
var pat_B = client.Read<Patient>("Patient/32/_history/4");
```

Tip: See the *paragraph about ResourceIdentity* for methods to construct URIs from separate values and other neat helper methods.

Note that `Read` can be used to get the most recent version of a resource as well as a specific version, and thus covers the two 'logical' REST interactions `read` and `vread`.

4.2.3 Updating a resource

Once you have retrieved a resource, you may edit its contents and send it back to the server. This is done using the `Update` interaction. It takes the resource instance previously retrieved as a parameter:

```
// Add a name to the patient, and update
pat_A.Name.Add(new HumanName().WithGiven("Christopher").AndFamily("Brown"));
var updated_pat = client.Update(pat_A);
```

There's always a chance that between retrieving the resource and sending an update, someone else has updated the resource as well. Servers supporting version-aware updates may refuse your update in this case and return a HTTP status code 409 (Conflict), which causes the `Update` interaction to throw a `FhirOperationException` with the same status code. Clients that are version-aware can indicate this using the optional second parameter `versionAware` set to `true`. This will result in a conditional call of the interaction.

4.2.4 Deleting a Resource

The `Delete` interaction on the FhirClient deletes a resource from the server. It is up to the server to decide whether the resource is actually removed from storage, or whether previous versions are still available for retrieval. The `Delete` interaction has multiple overloads to allow you to delete based on a url or a resource instance:

```
// Delete based on a url or resource location
var location = new Uri("http://server.fire.ly/Patient/33");
client.Delete(location);
// or
client.Delete("Patient/33");

// You may also delete based on an existing resource instance
client.Delete(pat_A);
```

The `Delete` interaction will fail and throw a `FhirOperationException` if the resource was already deleted or if the resource did not exist before deletion, and the server returned an error indicating that.

Note that sending an update to a resource after it has been deleted is not considered an error and may effectively "undelete" it.

4.3 Conditional interactions

The SDK provides support for the conditional versions of the `Create`, `Update` and `Delete` interactions. Not all servers will support conditional interactions and can return an HTTP 412 error with an OperationOutcome to indicate that.

All of the conditional interactions make use of search parameters. See the page of the resource type you want to work with in the HL7 FHIR specification to check which search parameters are available for that type. Then, setup the conditions.

For example, if we want to base the interaction on the `identifier` element of a resource, we can setup that search parameter with a value:

```
var conditions = new SearchParams();
conditions.Add("identifier", "http://ids.acme.org|123456");
```

Tip: See search for more explanation about `SearchParams` and example search syntax.

For the `Create` interaction you can have the server check if an equivalent resource already exists, based on the search parameters:

```
var created_pat_A = client.Create<Patient>(pat, conditions);
```

If no matches are found, the resource will be created. If one match is found, the server will not create the resource and will return an HTTP 200 (OK). In both cases `created_pat_A` will contain the resource that was sent back by the server, unless you set the FhirClient to ask for the minimal representation. When multiple resources match the conditions, the server will return an error.

To perform a conditional `Update`, the code is similar to that of the `Create` interaction above. Again, setup a `SearchParams` object and add it to your request:

```
// using the same conditions as in the previous example
var updated_pat_A = client.Update<Patient>(pat, conditions);
```

If a match is found, the update is performed on that match. If no matches are found, the server will perform the interaction as if it were a `Create`. When multiple resources match, the server will return an error.

The conditional `Delete` takes a string as first argument, indicating the resource type. The search parameters are passed as second argument:

```
client.Delete("Patient", conditions);
```

When no match is found, the server will return an error. If one match is found, that resource will be deleted. The server may choose to delete all resources if multiple instances match, or it may return an error.

4.4 Refreshing data

Whenever you have held a resource for some time, its data may have changed on the server because of changes made by others. At any time, you can refresh your local copy of the data by using the `Refresh` call, passing it the resource instance as returned by a previous `Read`, `Create`, or `Update`:

```
var refreshed_pat = client.Refresh(pat_A);
```

This call will go to the server and fetch the latest version and metadata of the resource as pointed to by the `Id` property in the resource instance passed as the parameter.

4.5 Message Handlers

The FhirClient provides you an option to add `HttpMessageHandlers`, which you can use to hook into the request/response cycle. With these handlers you can implement extra code to be executed right before a request is sent, or directly after a response has been received.

4.5.1 Adding extra headers

It could be necessary to add extra headers to the requests your FhirClient sends out, for example when you want to send an authorization token with your request. Or perhaps you need other code to be executed each time the FhirClient sends a request. This can be achieved by implementing your own message handler, for example an AuthorizationMessageHandler:

```
public class AuthorizationMessageHandler : HttpClientHandler
{
        public System.Net.Http.Headers.AuthenticationHeaderValue Authorization { get;␣
↪set; }
        protected async override Task<HttpResponseMessage>␣
↪SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
        {
                if (Authorization != null)
                        request.Headers.Authorization = Authorization;
                return await base.SendAsync(request, cancellationToken);
        }
}
```

and add that to the `FhirClient`:

```
var handler = new AuthorizationMessageHandler();
var bearerToken = "AbCdEf123456" //example-token;
handler.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);
var client = new FhirClient(handler);
client.Read<Patient>("example");
```

4.5.2 Chaining Multiple MessageHandlers

You can chain multiple `HttpMessageHandlers` as well to combine their functionality in a single FhirClient. You can do this using `DeligateHandlers`. DelegatingHandler is a handler that is designed to be chained with another handler, effectively forming a pipeline through which requests and responses will pass. Each handler has a chance to examine and/or modify the request before passing it to the next handler in the chain, and to examine and/or modify the response it receives from the next handler. Typically, the last handler in the pipeline is the `HttpClientHandler`, which communicates directly with the network.

For example, next to a AuthorizationMessageHandler, you might want to use both and a LoggingHandler:

```
public class LoggingHandler : DelegatingHandler
{
        private readonly ILogger _logger;

        public LoggingHandler(ILogger logger)
        {
                _logger = logger;
        }

        protected override async Task<HttpResponseMessage>␣
↪SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
        {
                _logger.Trace($"Request: {request}");
                try
                {
                        // base.SendAsync calls the inner handler
                        var response = await base.SendAsync(request,␣
↪cancellationToken);
```

(continues on next page)

(continued from previous page)

```
                        _logger.Trace($"Response: {response}");
                        return response;
                }
                catch (Exception ex)
                {
                        _logger.Error($"Failed to get response: {ex}");
                        throw;
                }
        }
}
```

source: https://thomaslevesque.com/2016/12/08/fun-with-the-httpclient-pipeline/

If you want to combine this with the AuthorizationMessageHandler from the previous section, you can add that to the LoggingHandler as an `InnerHeader`, because the LoggingHandler implements `DelegatingHandler`. Like this:

```
var authHandler = new AuthorizationMessageHandler();
var loggingHandler = new LoggingHandler()
{
        InnerHandler = authHandler
};
var client = new FhirClient("http://server.fire.ly", FhirClientSettings.
↪CreateDefault(), loggingHandler);
```

This puts the AuthorizationMessageHandler inside the LoggingHandler, which is added to the client. Resulting in that both handlers form a pipeline through which requests and responses will pass.

4.5.3 OnBeforeRequest and OnAfterResponse

To make use `OnBeforeRequest` and `OnAfterResponse` features that were on the previous implementation of the FhirClient, you can use the pre-defined `HttpClientEventHandler`. Use the `OnBeforeRequest` to add extra code before a request is executed by the FhirClient, and the `OnAfterResponse` event to add extra code that needs to be executed every time a response is received by the FhirClient:

```
using (var handler = new HttpClientEventHandler())
{
        using (FhirClient client = new FhirClient(testEndpoint, messageHandler:␣
↪handler))
        {
                handler.OnBeforeRequest += (sender, e) =>
                {
                        e.RawRequest.Headers.Authorization = new␣
↪AuthenticationHeaderValue("Bearer", "Your Oauth token");
                };

                handler.OnAfterResponse += (sender, e) =>
                {
                        Console.WriteLine("Received response with status: " + e.
↪RawResponse.StatusCode);
                };
        }
}
```

4.6 Retrieving resource history

There are several ways to retrieve version history for resources with the FhirClient.

Note: Servers are not required to support version retrieval. If the `history` interaction is supported, the server can can choose on which level it is supported. You can check the `CapabilityStatement` of the server to see what it supports.

4.6.1 History of a specific resource

The version history of a specific resource can be retrieved with the `History` method of the FhirClient. It is possible to specify a date, to include only the changes that were made after the given date, and a count to specify the maximum number of results returned.

The method returns a Bundle resource with the history for the resource instance, for example:

```
var pat_31_hist = client.History("Patient/31");
// or
var pat_31_hist = client.History("Patient/31", new FhirDateTime("2016-11-29").
↪ToDateTimeOffset());
// or
var pat_31_hist = client.History("Patient/31", new FhirDateTime("2016-11-29").
↪ToDateTimeOffset(), 5);
```

Note: The Bundle may contain entries without a resource, when the version of the instance was the result of a `delete` interaction.

4.6.2 History for a resource type

Sometimes you may want to retrieve the history for a **type** of resource instead of an instance (e.g. the versions of all Patients). In this case you can use the `TypeHistory` method.

```
var pat_hist = client.TypeHistory<Patient>();
```

As with the method on the instance level, a date and page size can optionally be specified.

4.6.3 System wide history

When a system wide history is needed, retrieving all versions of all resources, the FhirClient's `WholeSystemHistory` method is used. Again, it is possible to specify a date and a page size.

```
var lastMonth = DateTime.Today.AddMonths(-1);
var last_month_hist = client.WholeSystemHistory(since: lastMonth, pageSize: 10);
```

In this case the function retrieves all changes to all resources that have been done since the last month and limits the results to a maximum of 10. See *Paged results* for an example on how to page through the resulting Bundle.

4.7 Paged results

Normally, any FHIR server will limit the number of results returned for the *history* and *search* interactions. For these interactions you can also specify the maximum number of results you would want to receive client side.

The FhirClient has a `Continue` method to browse a search or history result Bundle, after the first page has been received. `Continue` supports a second parameter that allows you to set the direction in which you want to page: forward, backward, or directly to the first or last page of the result. The standard direction is to retrieve the next page. The method will return `null` when there is no link for the chosen direction in the Bundle you provide.

```
while( result != null )
{
    // Do something with the entries in the result Bundle

    // retrieve the next page of results
    result = client.Continue(result);
}

// go to the last page with the direction filled in:
var last_page = client.Continue(result, PageDirection.Last);
```

4.8 Searching for resources

FHIR has extensive support for searching resources through the use of the REST interface. Describing all the possibilities is outside the scope of this document, but much more details can be found online in the specification.

The FHIR client has a few operations to do basic search.

4.8.1 Searching within a specific type of resource

The most basic search is the client's `Search<T>(string[] criteria = null, string[] includes = null, int? pageSize = null)` function. It searches all resources of a specific type based on zero or more criteria. Criteria must conform to the parameters as they would be specified on the search URL in the REST interface, so for example searching for all patients named 'Eve' would look like this

```
Bundle results = client.Search<Patient>(new string[] { "family:exact=Eve" });
```

The search will return a Bundle containing entries for each resource found. It is even possible to leave out all criteria, effectively resulting in a search that returns all resources of the given type. Additionally, there is a `Search()` overload that does not use the generic `T` argument, you can pass the type of resource as a string in the first parameter instead.

4.8.2 Searching for a resource with a specific id

In some cases you may already have the id of a specific resource (e.g. an Observation with logical id `123`, corresponding to the url `Observation/123`). In this case you can use `SearchById<T>(string id, string[] includes = null, int? pageSize = null)`.

Note that this function still returns a Bundle. The operation differs from a `Read<T>()` operation because it can return *included* resources as well. E.g. given an id `123` for an Observation, you can ask a FHIR server to not only look for the indicated Observation but to return the associated `subject` as well:

```
var incl = new string[] { "Observation.subject" };
Bundle results = client.SearchById<Observation>("123", incl);
```

4.8.3 System wide search

Some servers allow you to execute searches across *all* resource types. This would use FhirClient's `WholeSystemSearch(string[] criteria = null, string[] includes = null, int? pageSize = null)`.

Doing this search:

```
Bundle results = client.WholeSystemSearch(new string[] { "name=foo" });
```

would then not only return Patients with "foo" in their name, but Devices named "foo" as well.

4.8.4 Complex searches

An alternative way to specify a query is by creating a `Query` resource and pass this to the client's `Search(Query q)` overload. The `Query` resource has a set of fluent calls to allow you to easily construct more complex queries:

```
 var q = new Query()
.For("Patient").Where("name:exact=ewout")
.OrderBy("birthDate", SortOrder.Descending)
.SummaryOnly().Include("Patient.managingOrganization")
.LimitTo(20);

Bundle result = client.Search(q);
```

Note that unlike the search options shown before, you can specify search ordering and the use of a summary result. As well, this syntax avoids the need to create arrays of strings as parameters and tends to be more readable.

4.8.5 Paged Results

Normally, any FHIR server will limit the number of search results returned. In the previous example, we explicitly limited the number of results per page to 20.

The FhirClient has a `Continue` function to browse a search result after the first page has been received using a `Search`:

```
var result = client.Search(q);

while( result != null )
{
    // Do something useful
    result = client.Continue(result);
}
```

Note that `Continue` supports a second parameter that allows you to browse forward, backward, or go immediately to the first or last page of the search result.

4.9 About resource identity

`Read` only takes urls as parameters, so if you have the Resource type and its Id as distinct data variables, use `ResourceIdentity`:

```
var patResultC = client.Read<Patient>(ResourceIdentity.Build("Patient","33"));
```

CHAPTER

FIVE

PARSING AND SERIALIZATION

The Firely .NET SDK makes it easy to work with XML and Json-based FHIR data. There are two approaches for getting data in and out of your application:

- Work with the POCO classes (as described in *Working with the model*). These are .NET classes that represent the FHIR resources mostly one-on-one.
- Work with the `ElementModel` classes, which is an abstract memory model representing the FHIR data as an in-memory tree of data.

The first approach is the simplest and is most applicable if you prefer working with strongly typed classes that align with the FHIR resources. E.g. there is a class `Patient` with a property `name`, as you would expect from looking at the FHIR documentation. For most users, this is all they need.

However, there are several reasons why the POCO-based approach may not work for you:

- The generated POCO classes are based on a specific version of FHIR, so if you need to deal with FHIR data independent of versions, POCO's are cumbersome to work with.
- The parsers for POCO classes cannot deal with incorrect data - there is no way to express invalid FHIR data as a POCO, so if you will get parser errors, there is no way to recover or correct the data.
- You may only be working with FHIR data in the sense that you need to be able to parse, persist and retrieve bits of data, without needing the full overhead of creating POCOs in memory.
- You need to be able to customize data when parsing or serializing data.

The second approach, using the `ElementModel` abstraction, has been designed to work with these usecases. In fact, the POCO parsers are built on top of the more low-level `ElementModel` classes.

5.1 Parsing with POCOs

Start by add this `using` directive to your code:

```
using Hl7.Fhir.Serialization;
```

You will now have access to the parsers and serializers for XML and Json:

- For XML: `FhirXmlParser` and `FhirXmlSerializer`
- For Json: `FhirJsonParser` and `FhirJsonSerializer`.

The way you would work with these does not differ much between XML or Json, so this section will just show you how to work with XML.

First, let us parse a bit of XML representing a FHIR Patient into the corresponding `Patient` class:

```
var xml = "<Patient xmlns='http://hl7.org/fhir'><active value='true'/></Patient>";
var parser = new FhirXmlParser();

try
{
    var parsedPatient = parser.Parse<Patient>(xml);
    Console.WriteLine(parsedPatient.Active);
}
catch (FormatException fe)
{
    // the boring stuff
}
```

In the example above, we knew the data contained a patient, but it is perfectly alright to be less specific and work with the `Resource` base class instead:

```
Resource parsedResource = parser.Parse<Resource>(xml);
```

You can then use C# constructions like the `is` operator to make your code behave differently depending on the type of resource parsed.

The `Parse` method has a few overloads, one of which allows you to pass in an `XmlReader` instead of a string, which makes sense if you have a stream of data that you don't want to read into a string first.

5.1.1 POCO's and parsing incorrect data

The POCO parsers are pretty strict about what data they will accept: since the data read and parsed must fit the POCO structure there is little room in allowing incorrect FHIR data. It is possible to allow a bit of flexibility however, which is controlled by passing a *ParserSettings* instance to the constructor of the xml or json parser:

```
var parser = new FhirXmlParser(new ParserSettings { AcceptUnknownMembers = true,
    AllowUnrecognizedEnums = true });
```

`AcceptUnknownMembers` will ensure the parser does not throw an exception when an instance of FHIR data contains an unknown (or incorrectly spelled) property. This is particularly useful when you want to make sure that your software will be able to read data from newer FHIR versions: for normative parts of the FHIR specification, all existing properties will remain unchanged, but newer versions of FHIR might add new members. By settings this property to `true`, you can make sure your older software can still read newer data. There is, however, no way to access the unrecognized data.

The same is true for `AllowUnrecognizedEnums`. When the parser turns a coded value (say Patient.gender) into an `enum`, the parser will allow values that are not part of the enumeration, and can therefore not be turned into an enumerated value. This means that the property will return a null value - you can, however, get to the 'unrecognized' value using the `ObjectValue` backing field, as demonstrated in the code below:

```
var parser = new FhirXmlParser(new ParserSettings {
    AllowUnrecognizedEnums = true });
p = parser.Parse<Patient>(xml2);    // xml2 contains a Patient.gender of 'superman'
Assert.IsNull(p.Gender);   // p.Gender will now be null
Assert.AreEqual("superman", p.GenderElement.ObjectValue);    // but you can query the␣
↪backing value
```

5.2 Serialization with POCOs

Serialization, unsurprisingly, turns a given POCO back into Json or XML, and is handled by either `FhirXmlSerializer` or `FhirJsonSerializer`. Both classes have several methods to serialize the POCO into different forms:

- `SerializeToString`, `SerializeToBytes` - will turn the POCO into an XML/Json string or directly into a UTF-8 encoded byte representation.
- `Serialize` - writes the POCO to an XmlWriter
- `SerializeToDocument` - turns the POCO into an `XDocument` or `JObject`

Continuing the previous example, we can change some value in the parsed Patient and then serialize it back out:

```
parsedPatient.active = false;
var serializer = new FhirXmlSerializer();
var xmlText = serializer.SerializeToString(parsedPatient);
```

Note that creating a new `FhirXmlSerializer` (or `FhirXmlParser`) is cheap. The constructor for the `FhirXmlSerializer` and `FhirJsonSerializer` take a single parameter to change settings, most notably to 'pretty print' your output:

```
var serializer = new FhirJsonSerializer(new SerializerSettings() {
    Pretty = true
});
```

5.2.1 Summaries

The FHIR specification introduces several summary versions of resources. You can serialize a POCO into one of these summary forms by passing the `summary` parameter to any of the serialization methods described above:

```
var xml = serializer.SerializeToString(b, summary: Fhir.Rest.SummaryType.Text);
```

5.2.2 Convenience methods

Caution: This documentation describes features in a prelease of version 1.0 of the SDK. The documentation may be outdated and code examples may become incorrect.

Although the code examples above are simple enough, there is also a set of extension methods available on POCOs to make serialization even easier, without the need of explicitly creating a serializer:

Table 1: Serialization of POCOs to different outputs

Method	Output
ToJson()	string
ToJsonBytes()	byte[]
ToJObject()	JObject
WriteTo()	JsonWriter
ToXml()	string
ToXmlBytes()	byte[]
ToXDocument()	XDocument
WriteTo()	XmlWriter

5.3 Introduction to ElementModel

While most developers will be most comfortable with using .NET POCOs to work with FHIR data, the Firely .NET SDK itself largely uses the ElementModel classes to read and manipulate data. These classes enable the SDK to work independently of FHIR versions (e.g. STU3, R4, R5, etc), and can even work with incorrect data. In addition, these classes make it simple to traverse data and obtain type information about the data without the need for .NET reflection.

5.3.1 The ElementModel interfaces

The ElementModel namespace contains two interfaces that represent FHIR data as a tree of nodes, each node containing children, primitive data, or both. They are `ISourceNode` and `ITypedElement`. The former is an abstraction on top of the serialized formats (currently XML, Json and RDF), whereas the second represents the strongly typed logical FHIR data model. The parsing SDK has a low-level `ISourceNode` implementation for each serialization format. The SDK then allows you to turn an untyped, low-level tree represented by the `ISourceNode` interface into a `ITypedElement` based typed tree by adding type information to it. Both interfaces can be found in the `Hl7.Fhir.ElementModel` assembly and namespace.

The next sections will details how to use the `ISourceNode` based parsers in the SDK, how to add type information and serialize data back out using the `ITypedElement` based serializers.

5.4 Parsing with ISourceNode

This interface exposes serialization-level, untyped instance data of a single resource at a level that abstracts away the specific details of the underlying representation (e.g. xml and json) and is shown below:

```
interface ISourceNode
{
    string Name { get; }
    string Text { get; }
    string Location { get; }
    IEnumerable<ISourceNode> Children(string name = null);
}
```

The interface represents a single node in the tree. Each node has a name, and a location (as a dot separated list of paths leading to the current node). The property `Text` contains the node's primitive data (if any).

Note that the name of the node is often, but not always, the same as the name of the element in the FHIR specification. The two may differ if the element has a choice of types (e.g. `Observation.value`). In this case, the name of of the source node is suffixed by the type, exactly as it would be in the serialized form. For the root of the tree, the name is the same as the type of the resource present in the instance.

Navigation through the tree is done by enumerating the children using the `Children()` method, optionally filtering on the name of the children.

Below is a tree which shows the instance data for an example Observation:

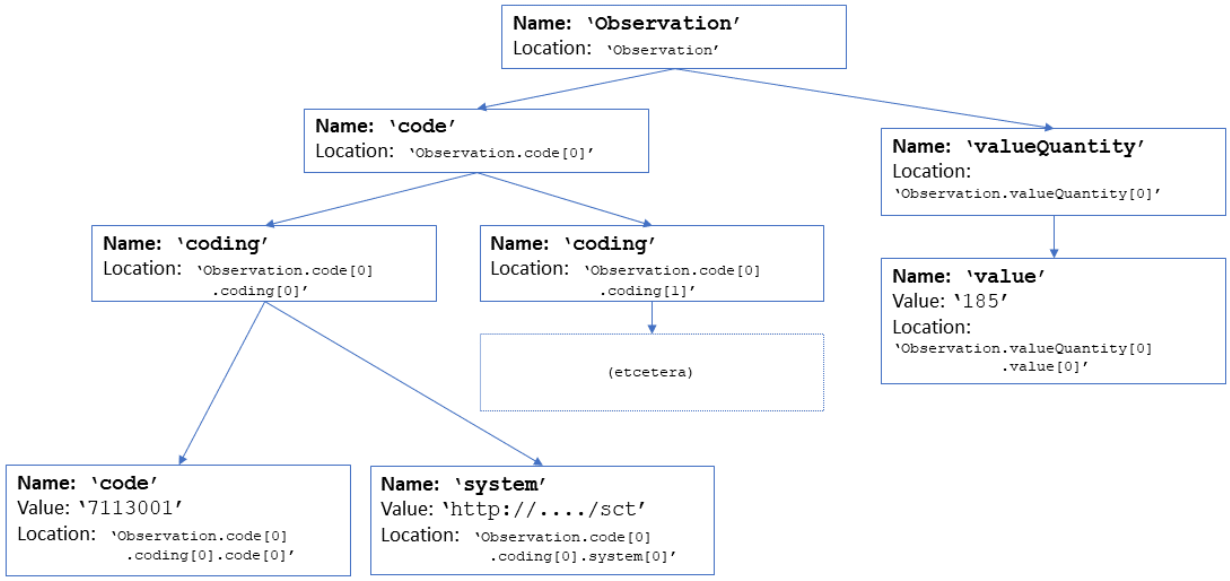
```
{
    "resourceType": "Observation",
    "code":
    {
        "coding": [ { "code": "7113001", "system": "http://snomed.info/sct" }, { "et"␣
↪: "cetera" } ]
    }
    "valueQuantity":
```

(continues on next page)

(continued from previous page)

```
    {
        "value": "185"
    }
}
```

The tree represented by the `ISourceNode` can be represented like this:



Some of the more subtle point illustrated here are:

- The root of the tree is named after the type of the instance (visible in Json as "resourceType", and a root element in Xml)
- There is no explicit representation of an array of repeated elements, arrays are flattened into sibling nodes with the same name, just like repeating elements in Xml.
- The location ensures each node is named uniquely by suffixing each path with an array index ([0], [1], etc), even if the given element does not repeat according to the FHIR specification (remember, at this level we do not have type information, nor are we aware of the differences between different versions of FHIR, so this information is simply not available).
- The choice element `value`, has its instance type (`Quantity`) appended to it: the name of the node in the tree agrees with the name of the element in the Json (or xml) serialization.

The SDK offers a set of extension methods on top of `ISourceNode` (like `Visit()` and `Descendants()`) to make it easier to select subtrees and process the data in the tree.

5.4.1 Parsing

The FHIR parsers available (currently for the FHIR Xml and Json formats) implement the `ISourceNode` interface and can be found in the `Hl7.Fhir.Serialization` assembly. The parsers are not created directly, instead there are two sets of factory methods, one for each serialization format: `FhrXmlNode` and `FhirJsonNode`. The factory methods are:

- `Read()`, to read data directly from an `XmlReader` or `JsonReader`.
- `Parse()`, to parse data from a string.
- `Create()`, to turn XElements, XDocuments or JObjects into an `ISourceNode`

All methods optionally take a `settings` parameter and return an `ISourceNode`, which reprents the root of the data read.

Here is an example parsing a string of xml and then quering some of its data:

```
var xml = "<Patient xmlns=\"http://hl7.org/fhir\"><identifier>" +
    "<use value=\"official\" /></identifier></Patient>";
var patientNode = FhirXmlNode.Parse(xml);
var use = patientNode.Children("identifier").Children("use").First();
Assert.AreEqual("official", use.Text);
Assert.AreEqual("Patient.identifier[0].use[0]", use.Location);
```

By swapping out the `FhirXmlNode` for an `FhirJsonNode` you can make this example to read Json data - there would not be any change to the rest of the code.

5.4.2 Constructing a tree in memory

It is also possible to construct an in-memory tree with data "by hand", using the `SourceNode` class. Since `SourceNode` implements `ISourceNode`, there would not be any difference from data read from a file or other source:

```
patient = SourceNode.Node("Patient",
    SourceNode.Resource("contained", "Observation", SourceNode.Valued("valueBoolean",
↪"true")),
    SourceNode.Valued("active", "true",
        annotatedNode,
        SourceNode.Valued("id", "myId2"),
        SourceNode.Node("extension",
            SourceNode.Valued("value", "4")),
        SourceNode.Node("extension",
            SourceNode.Valued("value", "world!"))));
```

Note that by using the C# `using static Hl7.Fhir.ElementModel.SourceNode;` this example could be make quite a bit shorter.

5.4.3 Handling parse errors

By default, parsing errors thrown as exceptions, but all parsers implement `IExceptionSource` to alter this behaviour. See *Handling errors* for more information.

The parsers try to parse the source *lazily*, so in order to detect all parse errors, one would have to do a complete visit of the tree, including forcing a read of the primitive data by getting the `Text` property. There is a convenience method `VisitAll()` that does exactly this. Additionally, there is a method `VisitAndCatch()` that will traverse the whole tree, returning a list of parsing errors and warnings.

5.5 Working with ITypedElement

The main difference between `ISourceNode` (see *Parsing with ISourceNode*) and `ITypedElement` is the presence of type information: metadata coming from the FHIR specification about which elements exist for a given version of FHIR, whether they repeat, what the type of each element is, etcetera. Type information is necessary for many operations on data, most notably serialization, running FhirPath statements and doing validation. As such, it is more common to work with `ITypedElement` than it is to work with `ISourceNode`. However, as one could imagine, in cases where type information is not necessary, `ISourceNode` is more performant and has a smaller memory footprint.

This is what `ITypedElement` looks like:

```
public interface ITypedElement
{
    string Name { get; }
    object Value { get; }
    string Location { get; }

    string InstanceType { get; }
    IElementDefinitionSummary Definition { get; }

    IEnumerable<ITypedElement> Children(string name=null);
}
```

Just like `ISourceNode`, the interface represents a single node in the tree, has a name and a location and you can enumerate its children using `Children()`.

Unlike `ISourceNode` however, type information is available by looking at the `InstanceType` and `Definition` properties. As a consequence, the `Name` property now returns the actually defined name of the element in the specification, so for choice types the element would not include the type suffix. For example, `Observation.valueQuantity` in an `ISourceNode` would turn into `Observation.value` in `ITypedElement`.

Similarly, the primitive value of the node (if any) is now of type object, and returns the value of the primitive, represented as a native .NET value. The following table lists the mapping between the encountered FHIR primitive and the .NET type used to represent the value:

Table 2: Mapping between FHIR type and .NET type

FHIR Type	.NET type
instant	Hl7.Fhir.ElementModel.Types.DateTime
time	Hl7.Fhir.ElementModel.Types.Time
date	Hl7.Fhir.ElementModel.Types.Date
dateTime	Hl7.Fhir.ElementModel.Types.DateTime
decimal	decimal
boolean	bool
integer	long
unsignedInt	long
positiveInt	long
string	string
code	string
id	string
uri, oid, uuid, canonical, url	string
markdown	string
base64Binary	string (uuencoded)
xhtml	string

Note that `Location` is exactly the same on both interfaces - every part of the path still has an array index, whether the element repeats or not. If you would like to have a shortened path, where non-repeating elements have their array index removed, you can check whether the underlying implementation of *ITypedElement* implements `IShortPathGenerator` (which the implementations from the SDK do), and get its `ShortPath` property.

Important: The `IElementDefinitionSummary` interface returned by the `Definition` property is very likely still to change. You are welcome to experiment with it and provide feedback, but the next release of the SDK will most likely add (incompatible) capabilities.

The SDK offers a set of extension methods on top of `ITypedElement` (like `Visit()` and `Descendants()`) to make it easier to select subtrees and process the data in the tree.

5.5.1 Obtaining an ITypedElement

The SDK enables you to turn data in POCO or `ISourceNode` form into an `ITypedElement` by calling the `ToTypedElement()` extension method.

In the first case, the POCO has all additional type information available (being based on a strongly-typed object model), and simply surface this through the `ITypedElement` interface. In the second case, the SDK needs an external source of type information to associate type information to the untyped nodes in the `ISourceNode` tree. The `ToTypedElement` method on `ISourceNode` looks like this:

```
public static ITypedElement ToTypedElement(this ISourceNode node,
    IStructureDefinitionSummaryProvider provider, string type = null,
    TypedElementSettings settings = null);
```

Notice that the `provider` parameter is used to pass in type information structured by the `IStructureDefinitionSummaryProvider` interface. Currently, the SDK supplies two implementations of this interface:

- The `PocoStructureDefinitionSummaryProvider`, which obtains type information from pre-compiled POCO classes. This is very similar to calling `ToTypedElement()` on a POCO, but this method does not require the caller to have data present in POCOs.

- The `StructureDefinitionSummaryProvider`, which obtains type information from `StructureDefinitions` provided with the core specification and additional Implementation Guides and packages. The constructor for this provider needs a reference to an `IResourceResolver`, which is the subsystem used to get access to FHIR's metadata resources (like `StructureDefinition`). See specification-sources for more information about `IResourceResolver`.

This is a complete example showing how to turn the `patientNode` from the last section into a `ITypedElement` by using external metadata providers:

```
ISourceNode patientNode = ...
IResourceResolver zipSource = ZipSource.CreateValidationSource();
ITypedElement patientRootElement = patientNode.ToTypedElement(zipSource);
ITypedElement activeElement = patientRootElement.Children("active").First();
Assert.AreEqual("boolean", activeElement.Type);
```

5.5.2 Compatibility with `IElementNavigator`

Previous versions of the SDK defined and used the precursor to `ITypedElement`, called `IElementNavigator`. Though functionally the same, `ITypedElement` is stateless, whereas `IElementNavigator` was not. To aid in parallellization, we have chosen to obsolete the stateful `IElementNavigator` in favor of `ITypedElement`. At this moment, not all parts of the SDK have been rewritten (yet) to use the new `ITypedElement` and we expect the same is true for current users of the SDK. To aid in migration from one concept to the other, the SDK provides a set of adapters to turn `IElementNavigators` into `ITypedElements` and vice versa. These can be constructed by simply calling `ToElementNavigator()` on a `ITypedElement` or `ToTypedElement()` on an `IElementNavigator`. The compiler will emit messages about this interface being obsolete to stimulate migration to the new paradigm.

5.5.3 Handling structural type errors

While traversing the ITypedElement tree, the implementations will try to associate type information from the specification with the data encountered. If this fails, errors are by default thrown as exceptions, but the all underlying implementations of ITypedElement implement `IExceptionSource` to alter this behaviour. See *Handling errors* for more information.

Detecting type errors is done *lazily*, so in order to detect all errors, one would have to do a complete visit of the tree, including forcing a read of the primitive data by getting the `Value` property. There is a convenience method `VisitAll()` that does exactly this. Additionally, there is a metehod `VisitAndCatch()` that will traverse the whole tree, returning a list of errors and warnings.

5.6 Serializing `ITypedElement` data

The SDK provides functionality to turn `ITypedElement` data into XML and JSON formats using the following set of extension methods:

Table 3: Serialization to different outputs

Method	Output
ToJson()	string
ToJsonBytes()	byte[]
ToJObject()	JObject
WriteTo()	JsonWriter
ToXml()	string
ToXmlBytes()	byte[]
ToXDocument()	XDocument
WriteTo()	XmlWriter
ToPoco()	Base
ToPoco<T>()	T (where T:Base)

The last two methods deserve a bit of explanation: from the standpoint of the serializers in the SDK, POCOs are also an output format. This means that in addition to the XML and JSON serializers, there are methods to turn `ITypedElement` data directly into POCO's. Continuing with last section's example:

```
ITypedElement patientRootElement = patientNode.ToTypedElement(zipSource);
string xml = patientRootElement.ToXml();
Patient p = patientRootElement.ToPoco<Patient>();
```

It will come as no surprise that the higher level serializers and parsers described in poco-parsing are thin convenience methods, simply calling the more primitive methods described in the last sections.

5.6.1 Working with subtrees

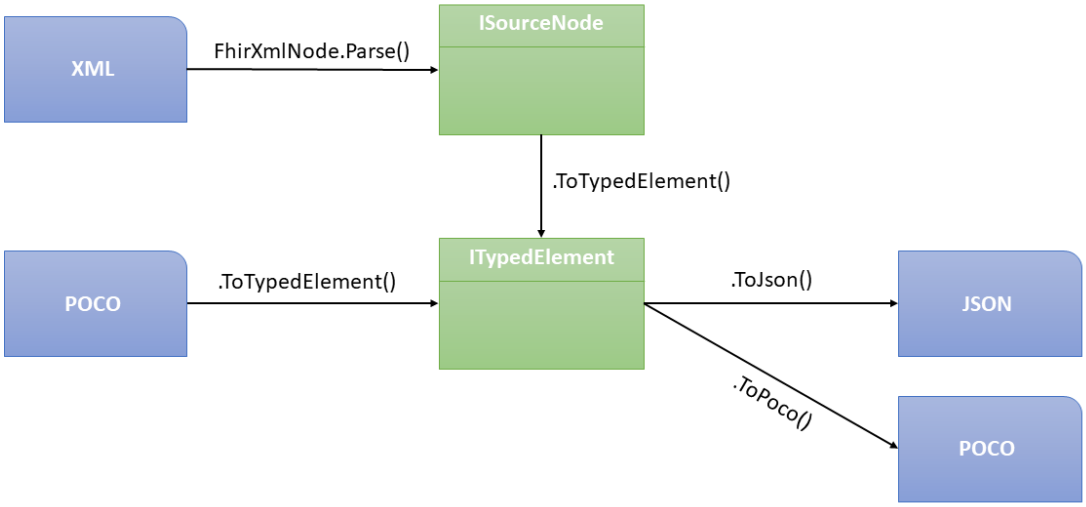
It is possible to traverse into a tree, away from the Resource root and then call one of the serialization methods above, with a caveat: the FHIR specification does not specify a "standard" serialization for subtrees, so there will not always be a "natural" way to serialize a subtree. More specifically, there is no way to serialize a subtree to JSON that represents a primitive, but this is quite possible in XML. The same is true for parsing into a subtree. The SDK will raise an error at runtime if the subtree cannot be represented in the chosen serialization.

5.6.2 Round-trip serialization

The FHIR serialization formats need type information to work correctly. For example, repeating elements require the use of an array in Json, while narrative in XML uses a different (xhtml) namespace. This is the reason that under most circumstances, serialization needs to be done based on the type-aware `ITypedElement` interface. However, when working with a single format (or doing a direct round-trip), the xml and json parsers annotate the (untyped) `ISourceNode` tree with enough information to allow for a serialization *without* type information, resulting in a cheaper round-trip than would be possible if the user has to retrieve and add type information first. As such, the serializers mentioned in this section have overloads on their methods to take a `ISourceNode` as well. This only works for round-trips however, so you will get a runtime exception when trying to serialize an `ISourceNode` to XML when it was created from JSON.

5.7 ElementModel Summary

The interfaces and methods described in this section are summarized in the picture below. Note that there are more conversions possible than shown (e.g. reading from an `XmlReader` is not explicitly depicted), and the SDK may be extended to support formats other than XML and JSON by the time you read this.



5.8 Writing your own `ITypedElement` implementation

The components that make up the parsing and serialization stack of the .NET framework are designed to be composed to be provide flexibility for different usecases and make it easy to add additional behaviour. You could build a pipeline converting XML to JSON by starting out with the *FhirXmlNode* (implementing *ISourceNode*), calling `ToTypedElement()` on it (effectively wrapping it in an internal implementation of *ITypedElement* called *TypedElement*). This is then again wrapped by the *FhirJsonBuilder* class, which turns the output into JSON.

This pipeline can be extended by custom implementations of both interfaces, and the SDK already contains a few extra's out of the box:

- The `MaskingNode`, which wraps another `ITypedElement` and which represents a tree pruned to just the nodes marked with isSummary. It could also be extended to mask out data based on the user's authorization.
- The `ScopedNode`, which tracks parent/child relationships and keeps track of the nearest "parent resource" (amonst other things) while traversing the tree. This information is used by both the validator and the FhirPath evaluation engine.

The MaskingNode source code is an excellent place to start exploring the possibilities provided by the framework.

5.9 Handling errors

It is pretty common to encounter errors while parsing (external) FHIR data - in fact so common that the Firely .NET SDK does not consider them true Exceptions but instead employs streaming error reporting for the parser classes. With streaming error reporting, you can subscribe to errors occuring in these components by installing a callback. When an error is encountered while parsing, this callback will be called with details about the error. There are several advantages to this approach:

- Since errors are common, we avoid the cost of raising a true .NET exception and the associated unwrapping of the stack.
- The parser will report errors as you navigate through the instance. This also means you can stop or halt wherever it fits your usecase.
- You are not limited to processing just the first error. As long as you continue navigating the instance, parsing will continue.

All components in the SDK (there are a few in addition to the parsers) that support streaming error reporting implement the `IExceptionSource` interface. It has a single property `ExceptionHandler`, which is a .NET delegate:

```
public delegate void ExceptionNotificationHandler(object source,␣
↪ExceptionNotification args);

public interface IExceptionSource
{
    ExceptionNotificationHandler ExceptionHandler { get; set; }
}
```

On encountering an error, such a component will invoke the delegate, passing it an `ExceptionNotification`, which is basically a wrapper around an (unraised) exception, with an additional message and severity.

Note that, if there is no such delegate installed, a component (in this case the parser) will just raise the exception instead.

5.9.1 Working with IExceptionSource

Installing a delegate by setting the `ExceptionHandler` property like this is feasible, but it is easy to forget to unregister the delegate (leading to unexpected call backs or memory leaks). In addition, you should check whether there was already a previously installed handler, which you to might need to forward the exception to once you have handled it.

To make working with `ExceptionHandler` easier, we have added a `Catch()` extension method to `IExceptionSource`, which returns an `IDisposable`, so you can use the returned value in a `using` statement like so:

```
List<Exception> allExceptions = new List<Exception>();
var src = FhirXmlNode.Parse("....");

using (src.Catch((_, arg) => allExceptions.Add(arg.Exception)))
{
    // navigate through src, or feed the instance to another
    // consumer, which will navigate it for you:
    var poco = src.ToPoco();
}
```

This will install your error callback, and uninstall it when the flow leaves the `using` block. `Catch()` has a `forward` argument which you can set to *true* to indicate whether you need to forward the errors to a handler that might have

been installed before calling `Catch()`.

If you are just interested in triggering all errors, you can do so by visiting the complete tree, catching the errors in the meantime:

```
using (src.Catch((_, arg) => allExceptions.Add(arg.Exception)))
{
    src.VisitAll();
}
```

or even more concise:

```
List<ExceptionNotification> errors = src.VisitAndCatch();
```

CHAPTER

SIX

CONTACT US

We actively monitor the issues coming in through the GitHub repository. You are welcome to register your bugs and feature suggestions there!

We are also present on chat.fhir.org on Zulip, on the dotnet stream, or for more general implementation questions, the implementers stream.

For broader discussions, there's a "FHIR on .NET" Google group.

If you're interested in a FHIR training, or want to know what other FHIR tools are available, please check here.

CHAPTER

SEVEN

RELEASE NOTES

Important: The release notes are moved to firely-net-sdk/releases.

The 2.0 transition docs can be found here.

7.1 1.3.0 (STU3, R4) (released 20190710)

7.1.1 Bugfixes

- 447/791 Disabled incorrect invariant rng-2 on Range datatype.
- 594 Suppressed useless information and warnings about non matching slices when slicing succeeds.
- 659 WebRequest task stays alive and holds connection after TimeoutException is thrown.
- 865 Validator accepted instant types without timezone
- 880 DirectorySource should not only detect but also report duplicate file names.
- 883 ZipSource is not handling files in subdirectories.
- 889 Annotated Additional Validation Rules would get confused by hybrid typed/untyped ITypedElement trees
- 895 DirectorySource should internally expand specified content directory to a full path to make matching reliable.
- 913 NavigatorStreamFactory (and thus DirectorySource) threw lots of caught exceptions, causing slow-down of directory scanning.
- 930 Summary and _elements now include non-mandatory children of mandatory top-level elements
- 933 Binding validation now returns success when validating a an non-bindeable instance
- 936 Internal ITypedElement implementation of Location property differed from the one in ElementNode.
- 941 The settings parameter was ignored for the xml and json serializer
- 965 Snapshotgenerator for R4 will now make typeref.(target)Profile replace base, not merge.
- 1002 Validation still enforced that contained resources cannot have text in R4.
- 1003 _sort did not allow the use of underscore search parameters.
- 1013 FhirClient threw incorrect kind of Exception when reading a non-FHIR body.

7.1.2 New Functionality

- 293 There is now a setting to ensure serialized files always end with newline
- 631 FhirClient gives a more informative error message when encountering an unparseable LastModified header
- 632 Improve validation of coded elements with null flavours
- 748/749 Expose ParserSettings and ExceptionNotificationHandler from Resource resolvers
- 750 Add ParserSettings to WebResolver
- 790/1016 ElementNode will now let you insert and delete nodes
- 890 NavigatorStream can now also return resources without an Id
- 892 ArtifactSummaryGenerator will now return error summaries together with file information
- 904 ArtifactSummary should indicate which entries are Bundles.
- 917 Validator now accepts custom resources
- 929 FhirClient now has support for "async" operations needed for bulk data.
- 938 Added factory methods on ElementNode to create primitive nodes implementing ITypedElement.
- 944 FhirUrl now implements IStringValue.
- 958 Allow Issue properties to be modified
- 959 FunctionCallExpression nodes can now be used without a Focus expression
- 992 Errors on FhirPath invariants now display structure definition and key for debugging purposes.
- 995 Re-enabled support for slicing by "pattern".

Note: Active development of DSTU2 has stopped, the included DSTU2 1.3 update is there to retain binary compatibility with the newer shared assemblies but has none of the new features.

7.2 1.2.1 (STU3, R4) (released 20190416)

The fix for issue #889 should have been in version 1.2.0, but it was not because of a merge error. In this hotfix (1.2.1) we corrected this mistake and added the fix for issue #889.

7.2.1 Bugfixes

- #889 Additional rules will get confused by hybrid typed trees

7.3 1.2.0 (DSTU2, STU3, R4) (released 20190329)

7.3.1 New Functionality

- Support for R4 final (4.0.0)
- #748 You can now set the ParserSettings used by the DirectorySource
- #756 Type-aware validations can now also be run by the POCO parser
- #924 Performance improvements

7.3.2 Bugfixes

- #907 Transaction code documentation was wrong
- #833 FhirClient.Read would return null when server not responding
- #896 StructureDefinitionElementDefinitionSummaryGenerator would not always derive IsRequired correctly
- #889 Additional rules will get confused by hybrid typed trees
- #888 TypedElementToSourceNodeAdapter should return ISourceNode annotation
- #872 Quantity now binds to <code>, not <unit>

Note: The parsing subsystem now catches way more error in the FHIR xml/json syntax. Since this would raise errors in instances that previously parsed without problems, this feature is turned off by default. Turn it on by setting ParserSettings.PermissiveParsing to "false".

7.4 1.2.0-beta2 (DSTU2, STU3, R4) (released 20190228)

Because of a mistake in creating the release of 1.2.0-beta1 for STU3, this beta1 release was not correct. The other 1.2.0-beta1 releases (DSTU2 and R4) were properly released. In STU3 you got a *System.MissingFieldException* during parsing a resource from xml to poco. This has been fixed now.

- Fix: Parsing of the primitive type decimal is done better now.

7.5 1.2.0-beta1 (DSTU2, STU3, R4) (released 20190220)

This first beta of 1.2 includes support for FHIR version R4! In the coming period we will investigate the performance of this release and take the last bugs out (if there are any :)). We will probably release a second beta, or if we are real confident about this beta1, we will create a final release 1.2.0.

- Fix: #750 The WebResolver should allow clients to configure custom ParserSettings for de-serialization
- Fix: #748 The DirectorySource should allow clients to configure ParserSettings for de-serialization
- Fix: #863 ParserSettings.Clone() does not copy PermissiveParsing
- #846 Enable cross-platform development
- Fix: #814 Empty contained block leads to *NullReferenceException*
- Fix: #824 Type-slice shortcut for choices in R4 does not yet work
- Added: #853 R4 Snapshot Generator
- Fix: #854 ElementDefinition.Base component #854
- Fix: #864 Validation of References to contained resources where there are multiple potential types, and the contained resource is not the first one targeted.
- Fix: #827 [Snapshot Generator] Child constraints are not included in snapshot when constraining a .value element
- Fix: #820 Fixed IgnoreUnknownElements naming inconsistency
- Improvement: #821 Better error reporting by always mentioning the context of the error
- Fix: #725 Debugger display doesn't work as the property has been renamed
- Fix: #817 Setting an empty Meta object breaks Json Serialization

- Fix: #756 Validation issue with text.div
- Fix: #807 FhirPath indexOf returns -1 instead of empty.

7.6 1.1.3 (DSTU2, STU3) (released 20190213)

Hotfix release

- Fix: ParserSettings.PermissiveParsing was not copied in Clone() and constructor

7.7 1.1.2 (DSTU2, STU3) (released 20190131)

Hotfix release

- Fix: v2-tables.xml of specification.zip contained an invalid codesystem (id =v2-0550)

7.8 1.1.1 (DSTU2, STU3) (released 20190130)

Hotfix release

- #817 Setting an empty Meta object breaks Json Serialization
- Fix: Added PermissiveParsing setting to ParserSettings
- Fix: Setting primitives to null AND using extensions would crash ITypedElement.Value

Note: Since 1.0, the parsers are more strict than before in what they accept as input. To disable this behaviour, set PermissiveParsing to 'true' in the ParserSettings which can be passed to the Json/Xml POCO parsers.

7.9 1.1.0 (DSTU2, STU3) (beta - final version to be released 20190128)

This is a minor release.

New functionality

- #180 We added a method to rerieve a contained resource on the parent resource by url.
- #460 Add support for _elements on the POCO serializer
- #731 FhirDateTime.ToDateTimeOffset() now asks you to pass in a TimeSpan - previously UTC was assumed.
- #578 DirectorySource will now resolve a specific version of a conformance resource if the canonical is versioned.
- #726 Add type-less overload for the POCO Parse() methods to mean "expect any resource type".
- #773 Added new FhirPath functions to support new invariants in R4.

Bugfixes

- #339 The Patient search parameter for some resources had an incorrect target list in ModelInfo
- #553 The namespace of the internal Sprache parser in the FhirPath assembly has been moved to a new namespace to avoid conflicts when also using the external assembly.
- #559 The LocalTerminologyService no longer throws an exception when there is no code at all passed to it.

- #561 The FhirPath compiler keeps a cache of recently encountered expressions. Multi-threaded access to it has been sped up.
- #624 Validator will now give a warning instead of an error if the display in a code is different from the one in the codesystem.
- #718 Corrected 'I' or "or" in invariants in DSTU2/STU3.
- #732 Json serialization no preserves significant digits.
- #746 ModelInfo.IsCoreModelTypeUri did not handle relative urls well.
- #754 Roundtripping patient.ToTypedElement().ToPoco<Patient>() failed.
- #755 Arguments to FhirClient.Search() with a key/value pair without value would throw an exception.
- #793 Element and Backbone element were handled differently in the ClassMappings that feed the parsers.
- #794 GetResourceFormatFromContentType would throw a null reference exception when the content-type had non-alphanumeric characters.

7.10 1.0.0 (DSTU2, STU3) (20181217)

This large release fixes about 80 issues - but more importantely introduces a completely new parsing/serialization subsystem that allows you to work without POCOs and also is more strictly following the serialization rules for XML and Json. This means you will get parse errors on instances that where (incorrectly) accepted as correct by the older versions of the API. More information on the new parsing subsystem can be found in the documentation. Please note that we have strived to keep the exsiting top-level POCO-parsing API intact - your projects should still compile without problems.

- #248 Json output can now be formatted
- #356 Parsing/serialization subsystem replaced to support working without using the generated POCO classes.
- #400 `TransactionBuiler.Delete` would cause "Invalid resource URL" in some circumstances.
- #433 Made the interface of all settings-related classes consistent across the whole API surface.
- #483 Introduction of `IErrorSource` to facilitate forwarding of errors and warnings between components of the API.
- #538 `Summary=true` still let some non-issummary fields through
- #569 Prefer header was not set on PUT
- #593 Fix .ttinclude file to prevent generating errors in some build environments.
- #619 Snapshot Generator ignores multiple codings with only display value
- #627 `ToFhirDateTime()` produced dateTimes without timezones when input DateTime.Kind was `Unspecified`
- #639 Target platforms are now 4.5, Netstandard 1.1, Netstandard 2.0
- #642 SnapshotGenerator does not expand custom element profile on Reference
- #657 Json Serializer was losing accuracy on serializing DatetimeOffset (last 4 digits in ticks)
- #663 Faster generation of property getters/setters with reflection emit
- #670 `DifferentialTreeConstructor` can now be used publicly to turn sparse differentials into full trees
- #676 Speed-up of serializers when running in Debug mode
- #684 `DirectorySource` can now retrieve summary data given a specific filename

- #696 `SummaryGenerator` now also extracts the extension context
- #704 Replaced uses of .NET `DateTime` with `DateTimeOffset` everywhere in the public API surface.
- Build scripts changed because of migration to Azure DevOps from AppVeyor

7.11 0.96.1 (Just R4) (released 20180925)

- R4-only release with all changes to the spec included upto the September ballot release.

Note: There are no new packages for other versions, since we did not add any new functionality.

7.12 0.96.0 (DSTU2, STU3 and R4) (released 20180606)

- #595 Added capability to harvest metadata directly from a stream
- #524 Search paths no longer use `[x]` suffix in DSTU2
- #556 Fixed threading issues in valueset expansion and snapshot generation when using the `CachedResolver`
- #577 `FhirBoolean.value` no longer has incorrect `[BooleanPattern]` in R4
- #591 Added `ignoreCase` option for parsing to enums
- #599 Fixed threading issue in FhirPath engine
- #601 Made the `DifferentialTreeConstructor` class public
- #606 FhirPath evaluator now support `Resource` and `DomainResource` as path roots
- #612 Bugfix for snapshot generator
- #614 Improved encoding for search parameters when doing POST-based search

7.13 0.95.0 (DSTU2, STU3 and R4) (released 20180412)

- Added support for R4 (warning: early alpha - client connectivity and parsing/serialization only)
- Added a SnapshotSource resource resolver that creates snapshots on the fly
- Added functionality to quickly harvest metadata from conformance resouces on a file system
- #423 Internal references within contained resources are not validated properly
- #474 Validation fails when start date is 0001-01-01
- #477 ZipSource is extracting the content in a temp directory without a discriminator for the spec version
- #479 Use search using POST
- #454 Invoking an operation using GET (i.e. $everything) does not work with primitive type as parameters.
- #494 Directory Source - single threaded by default
- #461 Check support for both types of extensions for regexes on primitive values (tracker GF#12665)
- #510 Missing diff annotation on ElementDefinition.TypeRefComponent
- #536 FhirClient.Transaction() method incorrectly POSTs to FHIR Base URl with trailing slash
- #544 Date.ToDateTime failed to convert “1976-12-12”

- #557 workaround for slice validation when discriminator is missing in slice
- #571 Serialize to XDocument and JObject directly

7.14 0.94.0 (DSTU2 and STU3) (released 20171207)

- #448, the FhirXmlSerializer/FhirJsonSerializer should now be instantiated, use of the static FhirSerializer class is obsoleted
- #434, the API is no longer creating empty <meta> tags in the serialization
- #420, the json parser no longer returns -1,-1 for positions on errors
- #412, added support for read-through and cache invalidation to CachedArtifactSource
- #355, the POCO parser is now using IElementNavigator as a source
- #474, fixed a bug where the parser would not accept '01-01-01' as a date
- #371, the validator will now complain when it encounters unsupported discriminator types
- #426, when you tell the validator to not follow external references, it will no longer produce warnings that it cannot locate the external references.
- #489, the validator would sometimes report incorrect indices in paths with repreating elements
- #477, the location where the specification.zip is unpacked now includes the version in the name, thus avoiding conflicts when switchin branches between dstu2/stu3 at development time
- #419, calling $everything with no parameters will no longer result in an incorrect http request.

7.15 0.92.5 (DSTU2) / 0.93.5 (STU3) (released 20171017)

Changes to both versions:

- Changed the `IElementNavigator` interface to enable skipping directly to a child with a given name, thus increasing navigation performance
- Improved performance of validation and fhirpath for POCOs
- Split off `IFhirClient` interface from the `FhirClient` implementation (primarily for testing/mocking)
- Many smaller bugfixes
- Improved error messages produced by the validator based on input from the NHS UK
- The validator will now let you put a constraint on children of `Resource.contained`, `Bundle.entry.resource` and similar nested resources.
- `SerializationUtil.XmlReaderFromString()` will no longer try to seek the stream passed in and rewind it.
- `TransactionBuilder` now has a (default) argument to specify the type of Bundle to build. Thanks mbaltus!
- `DirectorySource` now has Include/Exclude patterns (with globs) to have more control over directory scans for resource files.
- `DirectorySource` now supports processing conformance resources in json
- `FhirClient` now has async support

- You can now have `List<>` properties (like Extensions and other repeating elements) with null elements - these will simply be ignored and not serialized. Thanks wdebeau1!
- Made date to string and string to date conversion more consistent, fixing problems with locales using something else than ':' for time separators.
- Fixed an error where the `If-None-Exists` header included the base url of the server. Thanks angus-miller+tstolker!
- All `Search()` overloads on `FhirClient` now also have a `reverseInclude` parameter
- Update with a conditional would not set the `If-Match` header when doing a version-aware update. Thanks tstolker!
- `DeepCopy()` did not actually deep-copy collections - if you changed the original collection before you iterated over the clone, you would see the changes. This has been fixed. Thanks mattiasflodin!
- Client would not pass on 1xx and 3xx errors to client, instead throwing a generic `NotSupported` exception, making it harder to handle these errors by the client. Thanks tstolker!
- Added a fall-back terminology service so the validator can now invoke an external terminology service if the local in-memory service (provided with the API) fails.
- You can now specify a binding on an Extension, which translates to a binding on `Extension.value[x]`
- Fixed a bug where -if the definition of `element[x]` had a binding and a choice of bindeable and non-bindeable types- the validator would complain if the instance was actually a non-bindeable type.
- BREAKING: `FhirClientOperation.Operation` has been renamed to `RestOperation`
- BREAKING: Revision of calls to terminology services to support all parameters and overloads
- Validation across references will now include the path of the referring resource in errors about the referred resource to make interpretation of the outcomes easier.
- FhirPath's `resolve()` now actually works, and will resolve contained/bundled resources in the instance under evaluation. This also means the FhirPath evaluator will now take an EvaluationContext in which you can pass your resolver over to the evaluator.
- The enums in the generated code now also have an attribute on them with information about the codesystem, which can be retrieved using `GetSystem()` on any enum. Thanks brianpos!
- Added a few specific `[Serializable]` attributes to make the POCOs serializable with the Microsoft Orleans serializer. Thanks alexmarchis!
- Several improvements & bug fixes on the SnapshotGenerator
- Fixed handling of non-fhir json files in the conformance directory.
- Fixed `eld-16` constraint, which used an invalid regex escape `(\_)`
- Now using the new NuGet 3.3 `<contentFiles>` tag to replace the (failing) `install.ps1`, so a) you'll get the new `specification.zip` transitively in dependent projects and b) the build action will be correctly set.

DSTU2:

- Fixed small errors in the generated ConstraintComponent properties, giving more correct validation results

DSTU3:

- Fixes to the snapshot generator to create better ElementDefinition ids
- _sort parameter now uses STU3 format (_sort=a,-b,c) instead of modifier
- You can now set the preferred return to OperationOutcome. Thanks cknaap!
- You can now request the server to notify the client about unsupported search parameters. Thanks tstolker!

Changes to the DSTU2 version:

- Fixed small errors in the generated `ConstraintComponent` properties, giving more correct validation results

Changes to the STU3 version:

- Fixes to the snapshot generator to create better `ElementDefinition` ids
- `_sort` parameter now uses STU3 format `(_sort=a,-b,c)` instead of modifier
- You can now set the preferred return to `OperationOutcome`. Thanks cknaap!
- You can now request the server to notify the client about unsupported search parameters. Thanks tstolker!

7.16 0.90.6 (released 20160915)

- Fix: FhirClient will no longer always add *_summary=false* to search queries
- Fix: FhirClient will not throw parse errors anymore if the server indicated a non-success status (i.e. a 406)

7.17 0.90.5 (released 20160804)

- Enhancement: Portable45 target includes support for validation, and no longer depends on Silverlight 5 SDK. Thanks Tilo!
- Enhancement: Support for serialization where `_summary=data` (and automatically adds the Subsetted flag - temporarily adds the Tag then removes after serialization, if it wasn't there already)
- Enhancement: Added Debugger Displays for commonly used types
- Enhancement: Debugger Display for BundleEntries to show the HttpMethod and FullURL
- Enhancement: Additional method `public static bool IsKnownResource(FhirDefinedType type)` in ModelInfo (Thanks Marten)
- Enhancement: You can (and should) now create an instance of a `FhirXmlParser` or `FhirJsonParser` instead of using the static methods on `FhirParser`, so you can set error policies per instance.
- Enhancement: Introduced `ParserSettings` to configure parser on a per-instance basis:

```
FhirXmlParser parser = new FhirXmlParser(new ParserSettings { AcceptUnknownMembers =␣
↪true });
var patient = parser.Parse<Patient>(xmlWithPatientData);
```

- Enhancement: Introduced a setting to allow parser to parse (and serialize) unrecognized enumeration values. Use *Code<T>.ObjectValue* to get to get/set the string as it was encountered in the stream. The FhirClient now has a *ParserSettings* property to manage the parser used by the *FhirClient*.
- Enhancement: By popular demand: re-introduced *FhirClient.Refresh()*
- Enhancement: Snapshot generator now supports all DSTU2 features (re-slicing limited to extensions)

```
ArtifactResolver source = ArtifactResolver.CreateCachedDefault();
var settings = new SnapshotGeneratorSettings { IgnoreMissingTypeProfiles = true };
StructureDefinition profile;

var generator = new SnapshotGenerator(source, _settings);
generator.Generate(profile);
```

- Fix: Status 500 from a FHIR server with an HTML error message results in a FhirOperationException, not a FormatException. Thanks Tilo!
- Fix: *Code<T>* did not correctly implement *IsExactly()* and *Matches()*
- Fix: Now parses enumeration values with a member called "Equals" correctly.
- Fix: *Base.TypeName* would return incorrect name "Element" for Primitives and Code<T> (codes with enumerated values)
- And of course numerous bugfixes and code cleanups.

7.18 0.90.4 (released 20160105)

- Enhancement: Additional Extension methods for converting native types to/from FHIR types

```
public static DateTime? ToDateTime(this Model.FhirDateTime me)
public static DateTime? ToDateTime(this Model.Date me)
public static string ToFhirDate(this System.DateTime me)
public static string ToFhirDateTime(this System.DateTime me)
public static string ToFhirId(this System.Guid me)
```

- Enhancement: Added the *SnapshotGenerator* class to turn differential representations of a StructureDefinition into a snapshot. Note: we're still working with the Java and HAPI people to get the snapshots 100% compatible.
- Breaking change: All *BackboneElement* derived classes are now named as found on BackboneElement page in the specification, under the specializations heading. Usual fix for this will be removing the resource typename prefix from the classname, e.g. Bundle.BundleEntryComponent -> Bundle.EntryComponent
- Fix: Elements are not serialized correctly in summary mode
- Fix: Validate Operation does not work
- Fix: DeepCopy does not work on Careplan f201
- Fix: SearchParameters in ModelInfo are missing/have invalid Target values

From this version on, the model is now code generated using T4 templates within the build from the specification profile files (profiles-resources.xml, profiles-types.xml, search-parameters.xml and expansions.xml)

7.19 0.90.3 (released 20151201)

- Enhancement: IConformanceResource now also exposes the xxxElement members. Thanks, wmrutten!
- Enhancement: Parameters.GetSingleValue<> now accepts non-primtives as generic param. Thanks, yunwang!
- Enhancement: ContentType.GetResourceFormatFromContentType now supports charset information. Thanks, CorinaCiocanea!
- Enhancement: Operations can now be invoked using GET
- Fix: Small code analysis fixes. Thanks, bnantz!
- Fix: SearchParams now supports *_sort* without modifiers. Thanks, sunvenu!
- Fix: FhirClient: The "Prefer" header was never set. Thanks, CorinaCiocanea!
- Fix: FhirClient could not handle spurious OperationOutcome results on successful POST/PUT when Prefer=minimal. Thanks, CorinaCiocanea!

- Fix: Json serializer serialized decimal value "6" to "6.0". Thanks, CorinaCiocanea!
- Fix: Json serializer now retains full precision of decimal on roundtrip.
- Fix: ETag header was not correctly parsed. Thanks, CorinaCiocanea!
- Fix: Parameters with an "=" in the value (like pre-DSTU2 =<=) would become garbled when doing FhirClient.Continue(). Thanks rtaixghealth!
- Fix: FhirClient.Meta() operations will use GET and return Meta (not Parameters)

7.20 0.90.2

- Added support for $translate operations on ConceptMap
- Added support for the changed _summary parameter
- ArtifactResolver can now resolve ValueSets based on system
- The CachedArtifactSource is now thread-safe

7.21 0.90.0

- Updated the model to be compatible with DSTU2 (1.0.1)
- Added support for comments in Json
- Fixed a bug where elements called 'value' in Json could not have extensions or comments
- FhirClient now returns the status code in an OperationException
- Bugfixes

7.22 0.50.2

- Many bug and stability fixes
- ReturnFullResource will not only set the Prefer header, but will do a subsequent read if the server ignores the Prefer header.
- Client will accept 4xx and 5xx responses when the server does not return an OperationOutcome
- Client gives clearer errors when the server returns HTML instead of xml/json
- Call signatures for *OnBeforeRequest* and *OnAfterResponse* have been changed to give low-level access to body and native .NET objects. OnAfterResponse will now be called even if request failed or if response has parsing errors.
- The FhirClient has a full set of new LastXXX properties which return the last received status/resource/body.
- Serializers now correctly serialize the contents of a Bundle, even if summary=true

7.23 0.20.2

- FhirClient updated to handle conditional create/read/update, Preference header
- Introduction of TransactionBuilder class to easily compose Bundles containing transactions
- Model classes updated to the latest DSTU2 changes
- Serialization of extensions back to "DSTU1" style (as agreed in San Antonio)

7.24 0.20.1

- Added support for async

7.25 0.20.0

- This is the new DSTU2 release
- Supports the new DSTU2 resources and DSTU2 serialization
- Uses the new DSTU2 class hierarchy with Base, Resource, DomainResource and Bundle
- Further alignment between the Java RM and HAPI
- Support for using the DSTU2 Operation framework
- Many API improvements, including:
- deep compare (IsExactly) and deep copy (DeepCopy)
- Collections will be created on-demand, so you can just do patient.Name.Add() without having to set patient.Name to a collection first
- Note: support for .NET 4.0 has been dropped, we support .NET 4.5 and PCL 4.5

7.26 0.11.1

- Project now contains two assemblies: a "lightweight" core assembly (available across all platforms) and an additional library with profile and validation support.
- Added an XmlNs class with constants for all relevant xml namespaces used in FHIR
- Added *JsonXPathNavigator* to execute XPath statements over a FHIR-Json based document
- Added a new *Hl7.Fhir.Specification.Source* namespace that contains an *ArtifactResolver* class to obtain schema files, profiles and valuesets by uri or id. This class will read the provided validation.zip for the core artifacts. For more info see [here](artifacts.html).
- Changed *FhirUri* to use string internally, rather than the Uri class to guarantee round-trips and avoid url normalization issues
- All Resources and datatypes now support deep-copying using the *DeepCopy()* and *CopyTo()* methods.
- FhirClient supports *OnBeforeRequest* and *OnAfterRequest* hooks to enable the developer to plug in authentication.
- All primitives support *IsValidValue()* to check input against the constraints for FHIR primitives

- Models are up-to-date with FHIR 0.82
- And of course we fixed numerous bugs brought forward by the community

7.27 0.10.0

- There's a new *FhirParser.ParseQueryFromUriParameters()* function to parse URL parameters into a FHIR *Query* resource
- The Model classes now implements *INotifyPropertyChanged*
- FhirSerializer supports writing just the summary view of resources
- Model elements of type ResourceReference now have an additional *ReferencesAttribute* (metadata) that indicates the resource names a reference can point to
- ModelInfo now has information telling you which FHIR primitive types map to which .NET Model types (this only used to work for complex datatypes and resources before)
- We now support both .NET 4.0, .NET 4.5 and Portable Class Libraries 4.5
- For .NET 4.5, the FhirClient supports methods with the async signature
- All assemblies now have their associated xml documentation files bundled in the NuGet package
- Models are up-to-date with FHIR 0.80, DSTU build 2408

7.28 0.9.5

This release brings the .NET FHIR library up-to-date with the FHIR DSTU (0.8) version. Additionally, some major changes have been carried out:

- There is now *some* documentation
- The *FhirClient* calls have been changed after feedback of the early users. The most important changes are:
- The *Read()* call now accepts relative and absolute uri's as a parameter, so you can now do, say, a *Read(obs.subject.Reference)*. This means however that the old calling syntax like *Read("4")* cannot be used anymore, you need to pass at least a correct relative path like *Read("Patient/4")*.
- Since the FHIR *create* and *update* operations don't return a body anymore, by default the return value of *Create()* and *Update()* will be an empty *ResourceEntry*. If you specify the *refresh* parameter however, the FHIR client will immediately issue a read, to get the latest updated version from the server.
- The *Search()* signature has been simplified. You can now either use a very basic syntax (like *Search(new string[] {"name=john"})*), or switch to using the *Query* resource, which *Search()* now accepts as a (single) parameter as well.
- The validator has been renamed to *FhirValidator* and now behaves like the standard .NET validators: it validates one level deep only. To validate an object and it's children (e.g. a Bundle and all its entries and all its nested components and contained resources), specify the new *recursive* parameter.
- The validator will now validate the XHtml according to the restricted FHIR schema, so active content is disallowed.
- The library now *incorporates* the 0.8 version of the Resources. This means that developers using the API's source distribution need only to compile the project to have all necessary parts, there is no longer a dependency on the Model assembly compiled as part of publication. Note too that the distribution contains the 0.8 resources *only* (so, no more *Appointment* resources, etc.).

- The library no longer uses the .NET portable class libraries and is based on the normal .NET 4.0 profile. The portable class libraries proved still too unfinished to use comfortably. We've fallen back on conditional compiles for Windows Phone support. Cross-platform compilation has not been rigorously tested.
- After being updated continuously over the past two years, the FHIR client needed a big refactoring. The code should be readable again.

7.29 Before

Is history. If you really want, you can read the SVN and Git logs.

STU3	R4	R5

View our source code on Github

CHAPTER

EIGHT

WELCOME TO THE FIRELY .NET SDK'S DOCUMENTATION!

This is the documentation site for the support SDK for working with HL7 FHIR on the Microsoft .NET platform.

Important: The old name of this product was FHIR .NET API. Since November 2020 we renamed it to **Firely .NET SDK**. It is still the same product from the same contributors only with another name.

The library provides:

- Class models for working with the FHIR data model using POCO's
- A REST client for working with FHIR-compliant servers
- Xml and Json parsers and serializers
- Helper classes to work with the specification metadata, and generation of differentials
- Validator to validate instances against profiles
- A lightweight in-memory terminology server

On these pages we provide you with the documentation you need to get up and running with the SDK. We'll first explain how the FHIR model is represented in the SDK and give you code examples to work with the model. The FhirClient and its methods will also be demonstrated. Within an hour you can create your own simple FHIR client!

After those topics to get you started, we have added some pages that delve deeper into nice SDK features, such as parsing and serializing FHIR data, working with transactions, and using the ResourceIdentity functionality.

Note: All code examples on these pages are for the STU3 version of the library.

Please look at the *Contact us* page for ways to ask questions, contribute to the SDK, or reach out to other .Net developers in the FHIR community.