# CQL Engine Architecture

## Overview

This engine operates by consuming [ELM representation](https://cql.hl7.org/04-logicalspecification.html) of a CQL measure and takes the following steps:

* Transpile CQL (via ELM) to .NET expressions via the [System.Linq.Expressions](https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions?view=net-6.0) namespace
* Either:
  + Perform runtime compilation into invokable delegates
  + Translate these expressions into C# source code, which can then be compiled at leisure

This approach has a number of key advantages over engines which take the approach of simultaneously interpreting the ELM and producing the results. Chief among these is that once the transpiled .NET code has been compiled either into delegates or via source code first, the original ELM is no longer necessary, because it has now been transformed into executable code. This eliminates the need to continually examine the ELM for each execution. Secondarily, we are able to take advantage of the enormous body of work that has gone into optimizing .NET applications generally over the last twenty years. Although .NET code is not native and like Java runs on a virtual machine, this virtual machine is highly performant and pays only a very small penalty compared to pure native compilation (e.g., had we transpiled from CQL to ANSI C or C++ and compiled that instead).

As a result, this engine can perform exceptionally well when the source CQL is written with that concern in mind.

Further, once CQL libraries are compiled into measures, they can be referenced by regular .NET projects covering a wide range of use cases, e.g. a web service, console application, Azure Function, etc.

The two options outlined above are ideally suited for corresponding use cases:

* Runtime compilation when rapid iteration of the CQL is required – the authoring use case;
* Source code generation & compilation when the CQL is finished – the execution use case.

Source code generation is also preferable when debugging of the CQL is required. The resulting source code is transformed into a set of procedural steps with local variables defined, which allows line-by-line step-through.

## Key Projects & Classes

### ELM project

This project contains the domain objects to represent ELM as deserialized from JSON, and companion serialization helpers. **Do not try to deserialize using default mechanisms,** e.g.JsonConvert.Deserialize. Use the helper methods provided in the ElmPackage class.

#### ElmPackage class

This is the root class for an ELM json document and has static methods used to deserialize ELM from a variety of sources (disk, JSON string, Stream). Use the appropriate overload to acquire the correct object model.

#### Expression class

This is the root expression class in ELM that contains common properties (localId, locator, etc.) Note that during deserialization, the proper derivation will be picked based on the type property of the Expression.

### MeasureCompiler project

#### ExpressionBuilder class

This class is responsible for interpreting the ELM expression tree creating [LambdaExpression](https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.lambdaexpression?view=net-6.0) instances for all definitions and functions in the provided ELM package. This class relies on the OperatorBinding class defined in the CqlRuntime project to determine methods to call to handle CQL’s operators (e.g., adding two dates, checking whether an interval contains another interval, etc.). These operators are defined in an enumeration called CqlOperators.

#### TypeManager class

This class assists in determining which C# types to use for the CQL types named in the ELM. For standard well-known system types, it does this through the TypeResolver class in the CqlRuntime project, but for CQL tuples, the TypeManager is required to dynamically define new types with the members stipulated in the ELM.

#### SourceCodeWriter class

This class takes the output from the ExpressionBuilder’s Build method and translates them into C# source code. The writer relies on a number of [ExpressionVisitor](https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.expressionvisitor?view=net-6.0) implementations to create source code that is both readable and high performing, e.g., by declaring and deduping local variables. The SourceCodeWriter has room for improvement. Several optimizations we’ve considered remain unimplemented (e.g., short-circuiting is only experimental; both sides of conditionals are usually computed).

### CqlRuntime project

#### RuntimeContext class

An instance of this class is the only parameter to the LambdaExpressions built for CQL definitions, and it is the first parameter for the LambdaExpressions built for CQL functions, preceding all other function parameters.

Through this class, the LambdaExpressions can access the ICqlOperators instance that contains all the CQL operators.

**When operating without source code**, the Definitiions property **must be populated**. The LambdaExpressions that are built by the ExpressionBuilder call each other through the Definitions property. If for example your measure needs to call FHIRHelpers.”Normalize Interval” and the Definitions dictionary on the current RuntimeContext does not have a FHIRHelpers library or the “Normalize Interval” definition contained inside it, a cryptic runtime error will result.

You can translate a DefinitionDictionary of LambdaExpression to the required DefinitionDictionary of Delegate using the CompileAll() extension method. Delegate is the common base type of all Func<> instances, which are the result of calling the [standard .NET Compile method](https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.lambdaexpression.compile?view=net-6.0) on a LambdaExpression.

When operating with source code generated, this dictionary can be left empty as long as you are using the default ExpressionVisitors, which replace all references to Definitions with actual method calls e.g., your measure class will contain a private FHIRHelpers class member, and “Normalize Interval” will be invoked through that class member instead of through the Definitions dictionary.

The RuntimeContext contains an implementation of CqlComparers, which implements the ICqlComparer interface through matching the provided type with a dictionary of other ICqlCompares for specific types. CqlComparers holds an instance of IUnitConverter. RuntimeContext exposes the UnitConverter through accessing the CqlComparers instance.

The RuntimeContext also contains the patient bundle, typed as object. There are a handful of CQL operators that cannot be implemented without understanding the specific data model, e.g., to compute Age, the type of the Bundle containing the Patient resource must be known. It is expected that implementations of ICqlOperators will cast the Bundle member of the RuntimeContext appropriately.

The RuntimeContext contains all other input required to execute a measure: the value sets as an IValueSetDictionary, the parameters dictionary (string to object), and the date specified to be Now.

It is expected that **one RuntimeContext be created per bundle.** You should re-use the implementations mentioned above (e.g., Definitions, ICqlOperators, CqlComparers, IValueSetDictionary). Do not attempt to alter the Bundle property, even after all work is done. This may have unintended consequences and produce incorrect results.

#### CqlOperatorsBase class

This abstract class is the main functionality of the CQL engine. It implements ICqlOperators, which contains all of the methods that implement all of the operators outlined [here](https://cql.hl7.org/09-b-cqlreference.html).

**Operator methods are not virtual**. Methods are either abstract or instance methods, which means they cannot be overridden. This is by design. Virtual methods are slower to execute than non-virtual methods. If you need to modify the behavior of these operators, the recommended approach is to create a new implementation of ICqlOperators which composes an instance of CqlOperatorsBase and implements the interface through that instance. The C# compiler should aggressively inline those implementations since they are not virtual methods, and you will have the freedom to change whichever methods you wish.

#### ICqlOperatorsBinding class

This abstract class acts as the bridge between the ExpressionBuilder and the ICqlOperators interface. When the ExpressionBuilder interprets an ELM node that represents a system function (e.g., add two dates), it delegates the generation of the LINQ expression to a derivation of the OperatorBinding class, such as this one. The OperatorBinding’s job is to return an [Expression](https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.expression?view=net-6.0) instance that performs the operation requested by ELM. While this does not have to be done via an implementation of the ICqlOperators interface, it is recommended to do so as the RuntimeContext class exposes an ICqlOperators instance as a public member and thus will be available at runtime. Were you so inclined, you could for example instead bind to a static class whose operators are all static and take the RuntimeContext as a parameter.

This class is abstract because it does not implement Retrieve, the operator which is responsible for providing FHIR resources. It would not be possible to implement these methods without understanding the object model of the Bundle and its contained resources.

Implementing the Retrieve methods is how an implementation of the execution engine interfaces with FHIR data. We have implemented a minimal FHIR implementation including DTO classes for FHIR 4 and the required derivations of the abstract classes discussed in this document. That implementation implements its Retrieve methods by accessing properties of a Bundle instance provided with in the RuntimeContext derivation. That bundle is assumed to be the result of a [patient-everything](https://build.fhir.org/operation-patient-everything.html) API call. We expect the bundle to be available ahead of time for performance reasons, but another implementation could implement Retrieve by connecting to a FHIR server on-demand to get the resources requested.

We have also implemented a derivation that uses Firely’s data model for demonstration purposes but it is not as thoroughly tested as the simple model we created.

It would also be possible to create an ICqlOperators implementation that loads FHIR data from a non-standard data format (e.g., Parquet) or from a relational database. Note that CQL measures defined as “context Patient” measures expect that all resources returned from Retrieve methods will have a subject of one and only one patient for which the measure is being executed in context. Using an approach like this, you would need the RuntimeContext to have a patient ID property that you could use to formulate queries to match resources with that patient subject.

From a performance perspective, it is best to pre-fetch as much as possible all data and stage it as patient-everything bundles. Waiting for the result of a query on an external system for each Retrieve statement will result in sub-optimal performance. Retrieve methods are currently synchronous.

Most extant CQL, including CMS eCQMs and NCQA’s measures, are “context Patient” libraries.

#### CqlComparers class

This class acts as a dispatcher for other implementers of the ICqlComparer and IEquivalenceComparer interface. It maintains a dictionary keyed by the type of the left hand of the CQL operation. Generally, it is expected that attempting to compare unlike types is considered an error, so it is not required that a Date be explicitly comparable to a DateTime, for example.

The default comparers implemented in this project should suffice to cover CQL. As much as possible, CQL authors should avoid attempting to write comparisons that use any model types (e.g., FHIR.Date compared to System.Date). Implementers will be required to create those comparisons for their own model types, as the base engine does not know which .NET type you will use to model a FHIR.Date.

#### IUnitConverter interface & UnitConverter class

Unit conversion is a complex feature set that is easily an entire product on its own, so **the implementation provided here only handles the most common CQL unit use cases**, specifically converting time quantities (month to days, days to weeks, etc). If your measure domain will require unit conversion, it is recommended to provide your own implementation of this interface. Note that all units should be expressed as UCUM units; other unit code systems are not supported.

#### IValueSetDictionary interface & CqlValueSetDictionary class

This interface exposes two methods used by CQL to handle value set filtering in Retrieve statements, e.g. the [ResourceType : ValueSet] construct in CQL, as well as the ability for CQL to enumerate the codes in a value set. The provided implementation uses a hashing mechanism. **Value set URIs, code system URIs, and codes themselves are case-insensitive**. It is possible for a code system to define case-sensitive codes, but the default implementation provided does not support those, and it is not recommended to use case-sensitivity in code checking as it is highly error prone. The hashing mechanism ensures that determining whether a key is contained in a value set is an O(1) operation and is very fast.

#### TypeConverter class

The purpose of the TypeConverter class is to translate domain objects to and from CQL primitives used in the methods defined by ICqlOperators. For example, FHIR defines its own types for dates and times. Suppose your model provides a type called FhirDate and is used for Patient.birthDate. When Patient.birthDate is passed to a CQL function expecting a System.Date, a type conversion takes place. If the TypeConverter is not able to convert a FhirDate to a CqlDate (the primitive type which implements CQL’s System.Date type) then an exception will be thrown.

By default, TypeConverter provides very few conversions.

We implemented versions of Date, Time, and DateTime per the ISO8601 standard in the Iso8601 project(DateIso8601, TimeIso8601, and DateTimeIso8601, respectively). These types can be used to model FHIR’s date types if you so choose. Our Fhir.R4 package does use these Iso8601 dates, so we implemented the default TypeConverter class to be able to convert those types to CqlDate, CqlTime, and CqlDateTime.

#### TypeResolver class

This class helps the ExpressionBuilder, TypeManager, and OperatorBinding classes resolve types when crafting System.Linq.Expression instances.

This class is abstract, and must be implemented when adopting a model to use with the engine. When a CQL library declares that it is using FHIR, the CQL-to-ELM translator will allow you to use FHIR types by name. in the ELM, it will preface these detected FHIR types with “{http://hl7.org/fhir}”, e.g. “{http://hl7.org/fhir}Patient”. This type identifier string must be mapped to whatever class you wish to use to represent FHIR patient resources.

We provide a BaseTypeResolver class which provides mapping of the CQL system types to either C# primitives (e.g., "{urn:hl7-org:elm-types:r1}Integer" to int?). Remember that everything in CQL is nullable, so do not use non-nullable value types anywhere in this mapping or you will encounter errors.

#### CQL Primitives

The Ncqa.Cql.Runtime.Primitives namespace implements [the CQL system types](https://cql.hl7.org/09-b-cqlreference.html#types-2) where default C# types cannot be used, e.g. intervals, quantities, etc. Note that DateTimeOffset and TimeSpan cannot be used to capture dates and times in CQL because a great deal of focus in CQL is dealing with uncertainty related to precision; it is imperative to know exactly in which precision a DateTime is specified (e.g., down to the month, the hour, the millisecond, etc.). The .NET BCL DateTimeOffset type will fill with zeroes any value not provided when creating it, and will not track that it was created with only the year specified; this makes our precision comparisons impossible. We use custom ISO 8601 implementations for this located in the Iso8601 project.

#### DefinitionDictionary<T> class

This is a utility class that helps resolve function overloads (name + types of parameters) within libraries. The LambdaExpression instances created by ExpressionBuilder are stored in this structure, and when converted to Delegate instances for use by the RuntimeContext.

### Fhir project

#### Generator class

Located in the Ncqa.Fhir.Schemas namespace, this class loads structure definitions and interprets them to create C# source code which can then be compiled into a model. This is generally done only once, when a new set of structure definitions has been released by HL7, or to create a model for a specific IG.

This generator produces absolutely minimal DTOs by design. It uses literal property names following the HL7 names (e.g., “item” instead of the .NET standard Pascal-cased “Item”). This makes resolving properties on types much simpler for CQL expressions but is contrary to .NET design standards.

#### Serialization namespace

Contains several utility classes that are required when serializing/deserializing these models. FHIR JSON has the peculiar quirk of suffixing the element type to the property name (e.g., effectivePeriod vs. effectiveDateTime). Specific deserialization helpers are required to interpret these properly. These classes are built on Microsoft’s System.Text.Json package.

### Fhir.R4 project

This project contains the C# source code files generated by the schema Generator class for FHIR R4; these exist in the Model namespace.

#### FhirJson class

This static class provides Deserialize methods that use System.Text.Json’s JsonSerializer class primed with the required converters to handle FHIR’s JSON rules. **FhirJson.Deserialize** **must always be used** to deserialize these classes unless you want to provide your own custom converters to handle FHIR’s specific JSON rules.

### CqlRuntime.FhirR4 project

This project contains several classes that derive from the abstract types defined in the CqlRuntime project.

#### FhirCqlOperators

This derivation of CqlOperatorsBase implements Retrieve for known FHIR types. The default property paths (e.g. “code” for Procedure) [were taken from the model info used in MITRE’s reference engine](https://github.com/cqframework/cql-exec-fhir/blob/master/src/modelInfos/fhir-modelinfo-4.0.1.xml) and may have varying degrees of accuracy. Not all FHIR resources have default code paths, so not all can be used in Retrieve statements qualified with value set filters.

Retrieves are implemented by accessing the FhirRuntimeContext.Bundle property and pulling all resources of the specified type out of said bundle.

#### FhirOperatorsBinding

This derivation of ICqlOperatorsBinding implements Retrieve and binds method calls to FhirCqlOperators.

#### FhirTypeConverter

This derivation of TypeConverter adds type conversion for the Fhir.R4 model to the CQL primitive types and vice versa. This class is not fully implemented, so it is possible that some FHIR types are not mapped correctly.

#### FhirTypeResolver

This derivation of BaseTypeResolver probes attributes added to the generated types in the Ncqa.Fhir.R4.Model namespace to map the FHIR prefix used in CQL to the types in that namespace, e.g.:

[FhirUri("http://hl7.org/fhir/StructureDefinition/Claim")]

public partial class Claim : DomainResource

We read the tail path of the FhirUri attribute and thus will map this class to “{http://hl7.org/fhir}Claim”.

#### FhirCqlComparers

This derivation of CqlComparers adds comparers for model types from the Ncqa.Fhir.R4.Model namespace.

**Note that Resources are compared only by the value of its Id property, contrary to the letter of the specification**. This can be changed by setting the Comparers dictionary’s Resource key to a custom implementation that examines other properties. This is an optimization strategy that assumes all resources in a valid FHIR bundle have populated and distinct Ids. If you cannot make this guarantee, provide a custom implementation here.

The specification requires all Tuple types to be compared using Tuple comparison. The FHIR model’s resources are modeled as Tuple types from CQL’s perspective. Tuple comparison requires property-wise comparison, which is superfluous and expensive for large types such as FHIR resources.

#### FhirRuntimeContext

This derivation of RuntimeContext implements the abstract properties through strongly typed members (e.g., Operators through FhirOperators). This class must be used when providing a bundle modeled by the Ncqa.Fhir.R4 namespace.

## Extension scenarios

### How do I change the way two things are compared?

For example, suppose you want your FHIR resources to be compared not just by their [Id property](http://hl7.org/fhir/resource-definitions.html#Resource.id) but also values found in their [identifiers](http://hl7.org/fhir/patient-definitions.html#Patient.identifier) collection. To do this, you would implement the ICqlComparer interface for the Resource type with the desired behavior. After creating a RuntimeContext, you would access the context’s Comparers property to get a CqlComparers instance. You would then access that instance’s Comparers dictionary and replace the Resource key with an instance of your implementation. Now the engine will use your implementation to determine whether two resources are equal.

Remember that RuntimeContext is a per-bundle object. This means you must create a new RuntimeContext each time you run a new patient bundle through a measure, and assigning your custom Resource comparer must be done on each new RuntimeContext you instantiate.

### How do I change the way a CQL operator behaves?

The short answer is that you would implement ICqlOperators and set the Operators property of the RuntimeContext you are using with your implementation. Since RuntimeContext is abstract you will need to subclass RuntimeContext and initialize the Operators property in the constructor.

But generally you do not want to reimplement the entirety of CQL, so if you want to change only one operator, you would still do the above but you can implement ICqlOperators through a provided instance of CqlOperatorsBase, e.g.

public decimal? Abs(decimal? argument) => instance.Abs(argument);

Replace the body of the method(s) you wish to change with your own code. Visual Studio has an “implement interface through” option that will save you a lot of time if you want to go down this road.

### How do I call into a terminology service to check value sets?

Implement the IValueSetDictionary interface to contact your terminology service and pass this instance to the constructor of your RuntimeContext.

Note that the IsCodeInValueSet method **is called millions of times** in a typical CQL execution cycle, so this method **must perform extremely well**. The default implementation the engine requires that all codes be provided ahead of execution. Presuming that you have access to the ELM for the measure, you can “tree shake” the value sets referenced by your measure (and all of its dependencies) to determine which value sets you need to resolve from your terminology server, download them, and load them into the CqlValueSetDictionary implementation provided. This will ensure good runtime execution. You could also download each value set the first time it’s used, but this can lead to spikey execution.

At a minimum, you should implement a caching strategy. A disk caching strategy would be preferable. One of the pitfalls with measure execution is that results can change day to day if the external terminology server changes the way it expands value sets unbeknownst to the CQL engine. This can make troubleshooting issues unnecessarily difficult. Value sets should be considered as equal inputs to the engine as the measure ELM itself.

### How do I load FHIR data from a different data source?

The OperatorBinding class is abstract because it does not implement any Retrieve by default. Derivations of this class must implement this method by returning a MethodCallExpression. This controls how Retrieve constructs in the CQL, e.g. [Condition], are handled.

For example, you could create a DatabaseDataProvider class which implements a Retrieve method that connects to a database and executes a SQL query to get the resources with the specified type, code property, and value set constraint. You would then add an instance of this class to your RuntimeContext derivation. Your OperatorBinding derivation would then use the Expression.Call method to return a MethodCallExpression that invokes the RuntimeContext.DatabaseDataProvider.Retrieve method.

Using this technique you are not constrained even from loading FHIR JSON or XML. For example, you could also load C-CDA and then run it through a C-CDA-to-FHIR conversion before returning it to the engine.

### How do I use a different FHIR model implementation with this engine?

You would start by subclassing RuntimeContext and providing specific implementations of the following classes:

* CqlOperatorsBase (to implement Retrieve and Age calculations)
* ICqlOperatorsBinding (to bind the Retrieve operator to the new methods you implement in CqlOperatorsBase)
* TypeConverter (e.g. to translate your types into CQL system types, e.g. Period, Date, DateTime, etc.)
* BaseTypeResolver (to map FHIR types known to CQL to your corresponding types)
* CqlComparers (to register new comparers for your types, e.g. Resource, Coding, StringElement, etc)
* RuntimeContext (to tie all these implementations together).

This sounds like a lot of work, but each one of these classes is only a few lines of code.

### How do I use an entirely different model (e.g. OMOP, QDM) with this engine?

It is possible, but it is out of scope for this document; it is a significant amount of work that may require cooperation with the CQL-to-ELM tool.

### How do I call this engine from a different runtime (e.g. Java, NodeJS)?

We solved this problem internally by creating a small .NET daemon service that exposes a set of functions via gRPC that interact with the engine. We then use a gRPC client on the NodeJS side to call methods in the .NET code. There is very little overhead when using gRPC locally, especially using named pipes as a transport layer.

There are other techniques for in-process .NET interop, which will all work here. As long as you can call *any* .NET code from your process, you can use this engine.