Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Petr Onderka

.NET library for the MediaWiki API

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	Tomáš Petříček
Study programme:	Computer Science
Specialization:	General Computer Science

Prague 2012

Dedication.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: .NET knihovna pro MediaWiki API

Autor: Petr Onderka

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, University of Cambridge

Abstrakt:

Klíčová slova:

Title: .NET library for the MediaWiki API

Author: Petr Onderka

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Tomáš Petříček, University of Cambridge

Abstract:

Keywords:

Contents

Introduction

Wiki websites running on the MediaWiki software (such as Wikipedia) offer an API (Application programming interface) for programmatic access to their database. Since MediaWiki contains many functions, the API is quite extensive too: the core installation contains over seventy "modules" and more are available through extensions. Each module accepts parameters in the form of key-value pairs and returns a structured response in one of the possible formats, including XML.

Because of the size of the API, accessing it from some programming language is not very simple. Two basic approchaches are possible: static and dynamic.

The dynamic approach is to create a thin library around the API modules: let the user specify the names of the parameters and their types and return the response in a dynamic manner, possibly as an associative array, or something like XML DOM.

This way, the user is responsible for the correctness of his query and for correct processing of the response. Also, it is hard to discover what are the possible parameters, what values can they have and what form of response to expect. This can lead to excessive use of the documentation or a "trial and error" approach.

The static approach is to create an extensive library that contains methods tailored for every module, each returning a different statically-typed result.

This way, many of the errors the user could make will result in a compile-time error and the development environment can also advise the user what options are available.

But this approach is also inflexible: if the user wants to use something the library was not made for, he can't. Differences like this can be caused by different versions of the software, different sets of installed extensions, or just by different configuration.

Another question with the static approach is how to represent the parameters in code. Most modules have many optional parameters, and so presenting them to the user in an understandable manner might be a challenge.

One more problematic part might be how to represent choosing which properties to return in the result. A list of strings representing the chosen properties might be suitable for the dynamic approach, but not so much for the static one.

This work introduces the LinqToWiki library that tries to solve all those problems in the form of a .Net library.

The dynamic vs. static issues are solved by automatically generating statically typed code based on the metadata the API provides about itself. The code generation is performed using Roslyn, which is a new implementation of a compiler for the C# language written in C#.

The problems specific to the static approach are solved by using LINQ (Language integrated query): a set of features of the C# language and the .Net framework, that is useful for representing queries and their translation into another form.

The library also abstracts away some other aspects of the API, like paging of the results.

1. Problem analysis

The goal of the LinqToWiki library is to be able to express requests to the MediaWiki API in a way that is readable, discoverable and checked by the compiler for correctness as much as possible.

This is achieved by generating classes specific for each module and using them in LINQ queries.

The two most commonly used variants of LINQ are LINQ to Objects, and various versions of LINQ for SQL databases. LINQ to Objects is used for querying in-memory data, like arrays. There are several widely-used libraries for accessing SQL databases using LINQ, including LINQ to SQL, LINQ to Entities and NHibernate.

In all versions of LINQ, the queries look the same. For example:

```
from product in allProducts
where product.Price > 500
   && product.InStock
join category in categories on product.Category equals category
orderby product.Price
select product.Name
```

A query like this is translated into a sequence of method calls that take their parameters in the form of lambda expressions. For example, the **where** part of the above query is translated into:

```
Where(product => product.Price > 500 && product.InStock)
```

The commonalities between LINQ to Objects and SQL LINQ libraries are that the full range of operators is available and that all properties of the queried type are available in all of them.

The situation with the MediaWiki API is different in several ways:

1. It doesn't support queries represented by many of the LINQ operators, including **join** and **group by**.
2. Some of the modules don't support sorting, some do. Of those that do support sorting, some allow specifying the sort key, others only direction.
3. The sets of properties that are available for filtering, sorting and selecting are all different.
4. There are modules used for queries about a set of pages. Those pages can be from a hard-coded list or a result from some other module.
5. There are also parameters that don't fit into the LINQ model well. Some of them are required, some are not.

The goal is to be able to represent all valid queries, while invalid queries should cause a compile-time error.

Specifically, unsupported operators (like **join** and **group by**) should cause an error for all modules, while the **orderby** clause should cause an error only for the modules that don't support sorting.

Also, all operators should support only those properties that are actually supported by the API. So, for example for the `blocks` module, the following query should compile and execute fine:

```
from block in wiki.Blocks()
where block.Ip == "8.8.8.8"
orderby block descending
select block.ById
```

While the following query should cause three errors:

```
from block in wiki.Blocks()
where block.ById == 1234
orderby block.Expiry descending
select block.Ip
```

This is because limiting the query by the blocked IP address, sorting without specifying the key and selecting the ID of the user who performed the block are all allowed. On the other hand, limiting by the ID of the user who performed the block, sorting by the expiration date and selecting the IP address are all impossible. (Actually selecting the IP address of the blocked user is possible, but the information is contained in properties with different names.)

2. Background

2.1 MediaWiki API

MediaWiki is an open source wiki system. It is written in the PHP programming language and uses a relational database to store its data, usually MySQL. It is maintained by the Wikimedia Foundation, who also runs some of the biggest wikis, including Wikipedia and Wiktionary. It is also used by many others, including Wikia, who runs many small wikis for various interests and the unofficial wiki of the Faculty of Mathematics and Physics, ωικι.matfyz.cz.

There are several ways to programmatically access the database of some MediaWiki wiki. First, it's possible to directly access the database using SQL. This usually requires access to the server that runs the database, so it's not available in many cases. For Wikimedia wikis, a read-only access to most data, but excluding article texts, is available for registered users of Toolserver, run by Wikimedia Deutschland.

Mostly specific to Wikimedia wikis is also another option: database dumps. These are files that contain dumps of some tables of the wikis. Their disadvantages are that the newest dump is usually several days or weeks old and that the files can be huge, which is impractical for getting information about a small number of pages.

Last, but not least, is the MediaWiki API [1]. It can be used to remotely access any MediaWiki wiki (unless the API is disabled in the configuration) using the HTTP protocol.

Parameters for an API request are given in the query string of a GET request or in the body of a POST request (modules that perform modifications require the use of POST). The body of the POST request is usually formatted as `application/x-www-form-urlencoded`, but file uploads require the use of `multipart/form-data`.

Some parameters can accept multiple values at once. In these cases, the values are separated by a pipe character (`|`).

There are some parameters that are common to many modules:

- The `prop` parameter is used to determine what properties will be present in the response. The values of this parameter don't map directly to the properties of the response, so for example specifying `prop=ids` might cause the property `pageid` to appear in the result.
- The `dir` parameter is used to determine the order of the results: whether it should be ascending or descending.
- The `sort` parameter decides which property will be used to order the results of the query.

The response can be in one of the several available formats, the most widely used ones are XML and JSON.

The representation of most data types in the response is nothing unusual: `string`s are formatted as strings, `integer`s as decimal numbers, `timestamp`s are

formatted according to ISO 8601. Only `booleans` have a possibly unexpected representation: if the property is `false`, it is not present in the result at all, and if its value is `true`, it is represented as an empty string.

If there is some problem executing a request, for example if a parameter has an invalid value, a warning will be returned along with the result of the operation. In the case of a fatal problem, such as when the user doesn't have the right to perform an action, an error is returned, without any results.

The API is divided into modules and there are two kinds of modules: "normal" modules (called "non-query modules" in this work) and query modules.

Non-query modules are usually used to perform some action. For example the `edit` module can be used to edit a page and the `block` module can be used to block another user (it can be used only by users with sufficient privileges).

Query modules are used for retrieving information about the wiki. There are three types of query modules:

- `list` modules: Return contents of various lists. For example the `allcategories` module can be used to list all categories on a wiki, while the `categorymembers` module can be used to list members of a certain category.
- `prop` modules: Return information about a set of pages. For example, the `categories` module can be used to retrieve the categories for each page in a given set.
- `meta` modules: Return meta information that are not directly associated with pages. For example the `userinfo` module can be used to retrieve information about the currently logged-in user.

For `prop` modules, the set of pages they operate on can be specified directly using page titles or page IDs.

Another option is to use some other module (usually a `list` module) as a so called "generator". This way, one can for example retrieve all categories of pages in a specific category, by using the `categorymembers` module as a generator for the `categories` module.

Because more than one module can be used in one request, the parameters for each module are distinguished by using prefixes. For example, the prefix for the `categorymembers` module is `cm`. So, setting its `limit` parameter to the value of 5 can be achieved by adding `cmlimit=5` to the query string of a GET request or to the body of a POST request.

The API is also extensible: MediaWiki extensions can add their own modules and modify some behavior of existing modules.

An example of an API request URI and a response in the XML format is in Figure 2.1.

2.1.1 Paging

Because the results of the API queries can contain thousands and sometimes even millions of entries, the responses are limited. For most modules, the default limit (when it is not specified as a parameter) is ten entries per page and the default

```
http://en.wikipedia.org/w/api.php ? format = xml & action = query &
list = categorymembers & cmtitle = Category:Query%20languages &
cmprop = title & cmtype = page & cmdir = descending & cmlimit = 5
```

```
<?xml version="1.0"?>
<api>
  <query>
    <categorymembers>
      <cm ns="0" title="YQL (programming language)" />
      <cm ns="0" title="Yahoo! query language" />
      <cm ns="0" title="XQuery" />
      <cm ns="0" title="XPath" />
      <cm ns="0" title="XBase++" />
    </categorymembers>
  </query>
  <query-continue>
    <categorymembers cmcontinue="page|5842415345|572327" />
  </query-continue>
</api>
```

Figure 2.1: An example of an API request and a response

maximum is 500 entries for normal users. For users with the `apihighlimits` right, the limits are raised, usually to 5000 entries per page.

In the `limit` parameter, one can specify either the exact value, or the special value `max`, which means the maximum allowed for the current user.

To get the data from the following page, one has to use a value specified in the `query-continue` element in the result (see Figure 2.1 again). The value in this element is a transparent identifier of the next page.

The advantage of this system when compared with the conventional paging systems of numbering pages or using numeric offsets is that it avoids missing entries and duplicates when the result changes while retrieving the pages.

The API has no notion of transactions, so it is not possible to get fully consistent results that would correspond to an exact moment in time. But thanks to this paging system, one can be certain that an entry that should be in the result set during retrieving of all of the pages will actually be present in the result set exactly once.

The situation gets more complicated when using a `prop` module with another module as a generator. That is because both modules have their own paging.

When such a request is made, the first response will contain a limited number of items from the generator and a limited number of results from the `prop` module for those items. To retrieve the next set of items from the generator, one has to use the `query-continue` for the generator (called "primary paging" in this work). To retrieve the next set of results for the items from the first result, one has to use the `query-continue` for the `prop` module (called "secondary paging" here).

For an example, see Figure 2.2. It shows how the paging might work when using the `allpages` module as a generator, together with the `prop` module `categories`. The `query-continue` elements are not shown in the figure.

Figure 2.2: An example of primary and secondary paging

The situation is even more complicated with the `prop` module `revisions`. It can be used to retrieve information about revisions of pages, including their text and it is the only module that can be used to get the text of a set of pages.

For other modules, when no `limit` parameter is specified, a default value is used (usually 10) and a `query-continue` element is present in the response, to access the remaining items.

But for the `revisions` module, not specifying the `limit` parameter means that only the most recent revision will be shown and no `query-continue` will be present. Also, when `limit` is specified, the module can operate only on one page at a time, so for example one has to set the `limit` of a module used as a generator to 1.

2.1.2 The `paraminfo` module

A special importance for this work has the `meta` query module `paraminfo`. This module can be used to retrieve information about modules, which is necessary for generating code to access those modules in a static fashion.

Before this work, the `paraminfo` module provided some general information about the module and, most importantly, information about parameters, their data types and a short description, useful as a documentation for the generated code.

The data type of a parameter is either a simple type (e.g. `integer` or `string`), or an enumeration of possible values.

A shortened example of a response from the `paraminfo` module for the `categorymembers` module is in Figure 2.3.

For code generation in LinqToWiki, another piece of information is necessary: knowing the properties of the response and how do they map to the values of the `prop` parameter. For information about how we added them, see Chaper 3.

```
<module name="categorymembers" prefix="cm" querytype="list"
  generator="" listresult="" description="List all pages in a ...">
  <parameters>
    <param name="title" type="string"
      description="Which category to enumerate (required). ..." />
    <param name="pageid" type="integer"
      description="Page ID of the category to enumerate. ..." />
    <param name="prop" default="ids|title" multi=""
      description="What pieces of information to include ...">
      <type>
        <t>ids</t>
        <t>title</t>
        <t>sortkey</t>
        <t>sortkeyprefix</t>
        <t>type</t>
        <t>timestamp</t>
      </type>
    </param>
    <param name="namespace" multi="" type="namespace"
      description="Only include pages in these namespaces" />
    <param name="continue" type="string"
      description="For large categories, give the value ..." />
    <param name="limit" default="10" max="500" type="limit"
      description="The maximum number of pages to return." />
    <param name="sort" default="sortkey"
      description="Property to sort by">
      <type>
        <t>sortkey</t>
        <t>timestamp</t>
      </type>
    </param>
    <param name="dir" default="ascending"
      description="In which direction to sort">
      <type>
        <t>ascending</t>
        <t>descending</t>
      </type>
    </param>
  </parameters>
</module>
```

Figure 2.3: A shortened response of the `paraminfo` module for the `categorymembers` module

2.2 LINQ and expression trees

LINQ, short for Language Integrated Query, is a feature of the C# programming language[1] and the .Net framework that can be used for querying of various data sources and appeared in the version 3.0 of the language [2]. It uses higher-order functions and lambda expressions to achieve a readable declarative syntax.

LINQ consists of a set of so called "standard query operators": methods that are used to perform the query operations on a given source. Also, a special syntax (called "query expressions"), similar to SQL queries, is available for some of those operators. The compiler translates a query expression into a set of calls to standard query operators, using lambda expressions and anonymous types.

Anonymous types are types that don't have to be explicitly declared; they are used in similar situations as tuples in functional programming. An instance of an anonymous type is created by using the **new** keyword without specifying the type of the object to create.

For example the following query expression (as seen in Chapter 1):

```
from product in allProducts
where product.Price > 500
   && product.InStock
join category in categories on product.Category equals category
orderby product.Price
select product.Name
```

Is translated into the following method calls:

```
allProducts
    .Where(product => product.Price > 500 && product.InStock)
    .Join(
        categories,
        product => product.Category,
        category => category,
        (product, category) => new { product, category })
    .OrderBy(t => t.product.Price)
    .Select(t => t.product.Name)
```

The parameter `t` is called "transparent identifier". It is used to transfer a set of variables from one method call to another.

The LINQ library also contains methods that do not have a corresponding representation in query expressions. Some examples of those are `Aggregate()`, `Sum()` and `ToList()`.

Many of the basic query operators also correspond to a well-known higher-order functions from functional programming. See Figure 2.4 for comparison of some of the LINQ query operators, query expression clauses, and higher-order functions.

Usually, lambda expressions are compiled into normal methods and passed to the query operator methods as delegates (which are similar to function pointers in C or first-class functions in functional languages). But this would not be suitable for querying of sources that are not in-memory collections. This is because the

[1] Visual Basic .Net also supports LINQ, with slightly different syntax and capabilities, but uses the same types.

query operator	query expression clause	higher-order function
`Select()`	**`select`**, **`let`**	map
`Where()`	**`where`**	filter
`SelectMany()`	second and following **`from`**	bind
`Aggregate()`		fold
`Join()`	**`join`**	
`OrderBy()`, `OrderByDescending()`	**`orderby`**	
`GroupBy()`	**`group by`**	
`Sum()`		
`First()`		
`ToList()`		

Figure 2.4: Comparison between LINQ query operators, query expression clauses, and higher-order functions

query has to be translated into another form, like an SQL query or a set of parameters for the MediaWiki API.

Because of this, a lambda expression in C# can be also compiled into another form: an expression tree. Expression tree is an object that represents the given lambda expression in a form similar to an abstract syntax tree. This object can be programmatically accessed and manipulated, which allows translation of LINQ queries into other forms, such as SQL queries. An expression tree can also be compiled into a delegate.

For an example of an expression tree, see Figure 2.5.

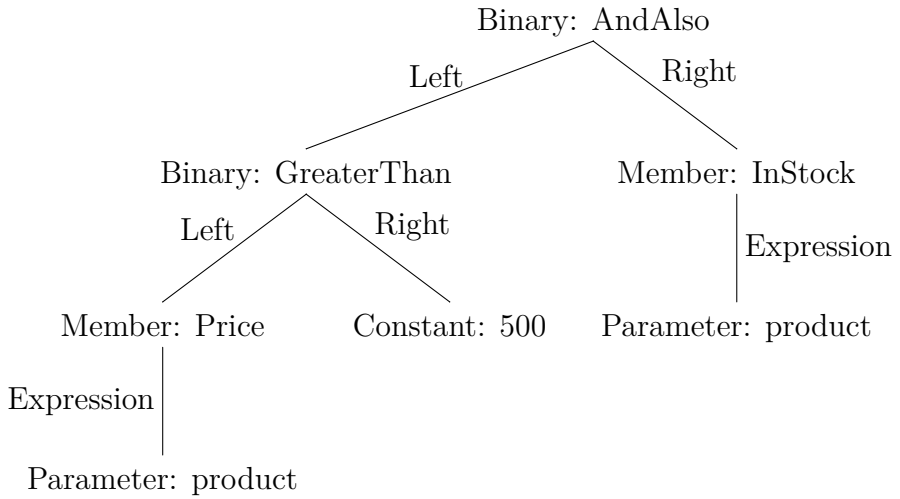

Figure 2.5: The body of the expression tree for the lambda expression `product => product.Price > 500 && product.InStock`

The .Net framework contains two implementations of the query expression pattern: the interfaces `IEnumerable<T>` and `IQueryable<T>`. This means that any object that implements one of these two interfaces can be used in a LINQ query.

These two types implement the query expression pattern completely, so they

can be used with any LINQ operator. Other custom types can implement only part of the query expression pattern, which would mean only a subset of the LINQ operators are available for such types.

The `IEnumerable<T>` interface usually represents an in-memory collection, so its implementation of the LINQ operators use delegates. The `IQueryable<T>` interface is usually used to represent a remote collection (such as a table in a relational database), so its version of the LINQ operators use expression trees.

The `IQueryable<T>` interface doesn't perform any translation of expression trees into the target query language. What it does is to combine the whole query into one expression tree, which is then passed to an implementation of `IQueryProvider`.

The query provider is then responsible for processing the expression tree and translating it into its target query language. If the query is not valid, the query provider will throw an exception at runtime.

2.3 Roslyn

Microsoft Roslyn is a new implementation of the C# compiler written in C# (and a VB.NET compiler written in VB.NET) [3]. Its main distinguishing characteristic is that it is "open": it can be used for example to convert between text and a syntax tree, to manipulate the syntax tree or to interrogate semantic information.

It also integrates itself into the Microsoft Visual Studio IDE (Integrated Development Environment), where it can be used to perform custom refactoring actions or to produce custom errors and warnings at compile-time.

Roslyn is currently under development and so far it had two public releases. Both were in the form of CTP (Community Technology Preview), the first one from October 2011, the second one from June 2012.

In the second CTP, the syntactic part of the library in completely implemented, so for example the syntax tree can represent any construct of C# and any syntax tree can be translated to and from source code. On the other hand, the semantic part of the library is not fully implemented, which means that for example some syntax trees won't successfully compile, even if they represent valid C# code.

Because of its close relation with Visual Studio, Roslyn syntax tree is able to represent every feature of C# with down to character precision. This includes "trivia": parts of code that are not significant for the compiler, such as whitespace and comments.

Trivia can also be "structured", that is, it can form a small syntax tree of its own. An example of structured trivia are XML documentation comments, that can be used to provide documentation for a piece of code, which can then be automatically processed.

For an example of a Roslyn syntax tree, see Figure 2.6

```
public abstract CategoryInfoResult CategoryInfo { get; }
```

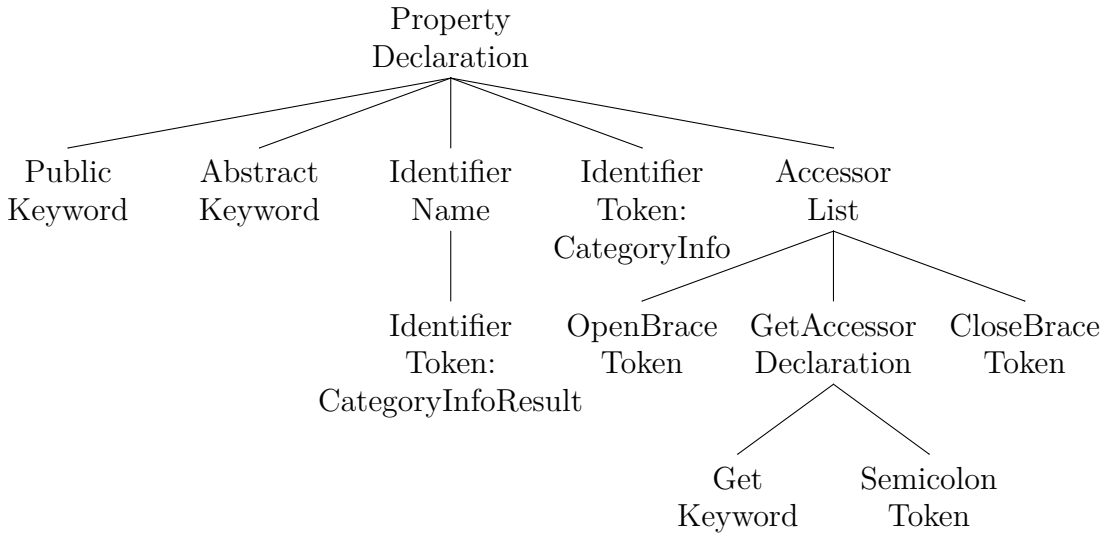

Figure 2.6: An example of a piece of C# code and its Roslyn syntax tree (trivia not shown)

Roslyn syntax trees are immutable and can be created using factory methods from the `Syntax` class. And while not all elements of the syntax tree have to be specified (like braces of a property accessor list), creating a syntax tree can be quite cumbersome.

The exact syntax for creating syntax trees changed between the two CTPs. In the October 2011 CTP, methods with many optional parameters were used. In the June 2012 CTP, the situation somewhat improved: the factory method now has parameters only for required children of the created node and optional child nodes can be added in a fluent manner using `With*` methods.

For an example of code to manually create the syntax tree from Figure 2.6, see Figure 2.7 for the October 2011 CTP version and Figure 2.8 for the June 2012 CTP version.

```
Syntax.PropertyDeclaration(
  modifiers:
    Syntax.TokenList(
      Syntax.Token(SyntaxKind.PublicKeyword),
      Syntax.Token(SyntaxKind.AbstractKeyword)),
  type: Syntax.ParseTypeName("CategoryInfoResult"),
  identifier: Syntax.Identifier("CategoryInfo"),
  accessorList:
    Syntax.AccessorList(
      accessors:
        Syntax.List(
          Syntax.AccessorDeclaration(
            SyntaxKind.GetAccessorDeclaration,
            semicolonTokenOpt:
              Syntax.Token(SyntaxKind.SemicolonToken)))));
```

Figure 2.7: A sample code to manually create a Roslyn syntax tree using October 2011 CTP

```
Syntax.PropertyDeclaration(
  Syntax.ParseTypeName("CategoryInfoResult"),
  "CategoryInfo")
  .WithModifiers(
    Syntax.TokenList(
      Syntax.Token(SyntaxKind.PublicKeyword),
      Syntax.Token(SyntaxKind.AbstractKeyword)))
  .WithAccessorList(
    Syntax.AccessorList(
      Syntax.List(
        Syntax.AccessorDeclaration(
          SyntaxKind.GetAccessorDeclaration)
          .WithSemicolonToken(
            Syntax.Token(SyntaxKind.SemicolonToken)))));
```

Figure 2.8: A sample code to manually create a Roslyn syntax tree using June 2012 CTP

3. MediaWiki improvements

As mentioned in Section 2.1.2, to generate types for each module of the API, it is necessary to know the properties contained in the module response and how do they map to the values of the `prop` parameter.

Because this information was not available, as a part of this work, the `paraminfo` module was extended to be able to provide information about result properties of the API modules, using the same type system already used to describe parameters. Also, most of the API modules were changed so that they can provide this information to the `paraminfo` module.

Of the 73 modules present in the MediaWiki core (that is, without any extensions), 5 are not suitable for having their result properties described, because their result looks different than the result of other modules (for example, there are modules that produce RSS feeds). Further 5 modules do use the same response format as the other modules, but their response cannot be described in the type system used. There are also 17 modules that can be partially represented using this type system, but not completely.

The patch that adds this ability to the `paraminfo` module and the necessary information to most other modules was reviewed by MediaWiki developers and merged into the official repository on 12 June. On 2 July, MediaWiki version 1.20wmf6, which includes changes from this patch, was deployed to all Wikimedia sites, including Wikipedias.

An example of the added result information to the `paraminfo` response (here for the `categorymembers` module) is in Figure 3.1.

During this work, we also noticed several bugs and inconsistencies in the API. Because of this, we reported eight bugs to the WikiMedia bug-tracking system. Three of them turned out to be duplicates of already reported bugs and, as of 10 May 2012, three of them are still waiting to be fixed.

We also submitted eight additional patches to the MediaWiki code review system. Although only three of them actually fix behavior of the MediaWiki API, the rest are only fixes in documentation and other mostly insignificant changes. Of those three patches, one is still waiting for review, because it is a breaking change, and most likely won't be accepted.

```
<props>
  <prop name="ids">
    <properties>
      <property name="pageid" type="integer" />
    </properties>
  </prop>
  <prop name="title">
    <properties>
      <property name="ns" type="namespace" />
      <property name="title" type="string" />
    </properties>
  </prop>
  <prop name="sortkey">
    <properties>
      <property name="sortkey" type="string" />
    </properties>
  </prop>
  <prop name="sortkeyprefix">
    <properties>
      <property name="sortkeyprefix" type="string" />
    </properties>
  </prop>
  <prop name="type">
    <properties>
      <property name="type">
      <type>
        <t>page</t>
        <t>subcat</t>
        <t>file</t>
      </type>
      </property>
    </properties>
  </prop>
  <prop name="timestamp">
    <properties>
      <property name="timestamp" type="timestamp" />
    </properties>
  </prop>
</props>
```

Figure 3.1: Result properties information for the `categorymembers` module

4. The LinqToWiki library

The LinqToWiki library consists of one Visual Studio solution, that contains the following projects:

- LinqToWiki.Core
- LinqToWiki.Codegen
- LinqToWiki.Codegen.App
- LinqToWiki.ManuallyGenerated
- LinqToWiki.Samples

The LinqToWiki.Core project contains the core of the library: types that access the API, convert to and from the representation of data in the API, represent parameters of various types of queries, represent query results or those that process LINQ expression trees. This project can be used together with code generated using LinqToWiki.Codegen, or with manually written code.

The LinqToWiki.Codegen project handles generating code based on information from the `paraminfo` module. It contains types that represent the results of that module, process them, generate C# code and compile this code. This project also contains helper types for easier creating of Roslyn syntax trees.

The LinqToWiki.Codegen.App project compiles down to a simple console application called linqtowiki-codegen, that uses functionality from the LinqToWiki.Codegen project.

The LinqToWiki.ManuallyGenerated project is a sample of how one could write code to access a wiki using LinqToWiki without using LinqToWiki.Codegen to generate the code.

Finally, the LinqToWiki.Samples project contains samples showing how to use various API modules using LinqToWiki. It uses code generated by LinqToWiki.Codegen.App.

The intended usage of LinqToWiki is this: First run the linqtowiki-codegen application to generate a DLL library tailored for a certain wiki. Then use the generated library together with LinqToWiki.Core in your C# (or VB.NET) application to access that wiki.

Other options are possible, though. For example, the LinqToWiki.Codegen library can be used to generate the code as a set of files containing C# source code. Those files can then be modified and manually compiled.

4.1 The LinqToWiki.Core project

The LinqToWiki.Core project contains shared code that can be used when querying any MediaWiki wiki that has the API enabled. It can be used together with code generated through LinqToWiki.Codegen, but it can also be used without it.

In fact, LinqToWiki.Codegen internally uses LinqToWiki.Core to access the `paraminfo` module using manually written code.

4.1.1 `QueryTypeProperties`

The `QueryTypeProperties` class holds basic information about a "query type", which corresponds to an API module. This information includes the prefix this module uses in its parameters, what type of module it is or mapping of its result properties to values accepted by the `prop` parameter. It is also able to parse XML elements this module returns.

4.1.2 `WikiQuery`

Probably the most often used and certainly the most interesting queries are those using `list` query modules. Such queries are represented in LinqToWiki by a group of types whose names start with `WikiQuery`.

Specifically, there are four such types: `WikiQuery`, `WikiQuerySortable`, `WikiQueryGenerator` and `WikiQuerySortableGenerator`. If a module supports sorting, it is represented by a type with `Sortable` in its name and if it supports being used as a generator for `prop` queries, it is represented by a type with `Generator` in its name.

There is also a fifth type: `WikiQueryResult`. This type by itself represents a query that can't be modified anymore, but can be used to execute it and get the results. All of the four preceding types inherit from `WikiQueryResult`, so it is possible to execute the query using any one of them too.

The type governs what operations are available. For example, if a type is one of the two `Sortable` types, it will have an `OrderBy()` method, but no other type has this method. Each method can also return a different type, as is necessary to form queries.

All of the `WikiQuery`-related types are generic and their type parameters are used to decide what properties can be used in each operation. For example, the type parameter `TOrderBy` of `WikiQuerySortable` decides what properties can be used in the parameter of the `OrderBy` method.

The way this is achieved is that `TOrderBy` is a type that contains the properties that can be used for sorting in the module `WikiQuerySortable` represents and the `OrderBy` method accepts lambda expressions whose parameter is of this type.

For example, if some module supported sorting by `PageId` and `Title`, then `TOrderBy` would be a type that contains two properties with those names. Because of this, a query like `source.OrderBy(x => x.Title)` would compile and execute fine, but `source.OrderBy(x => x.Name)` would fail to compile.

Because of the way lambda expressions work, queries like `source.OrderBy(x => x.Title.Substring(1))` or `source.OrderBy(x => random.Next())` would

compile fine. But because there is no way to efficiently execute such queries using the MediaWiki API, they will fail with an exception at runtime.

The various methods available on the `WikiQuery` types are:

- `Where()` only sets some parameter or parameters of a query, it always returns the same type.

  It is available on all four of the basic `WikiQuery` types and uses the generic type parameter `TWhere`.

- `Select()` is used to choose how the elements in the resulting collection should look like and what properties should they contain. Because the result of the lambda passed into this method can be an arbitrary type, it doesn't make sense to modify the query after calling this method. Because of that, `Select()` returns `WikiQueryResult`. This also follows query expression syntax, where **`select`** is the last clause of each query.

  It is available on all four of the `WikiQuery` types and uses the type parameter `TSelect`.

- `ToEnumerable()` and `ToList()` are used to actually execute the query. The distinction between the two methods is that `ToEnumerable()` returns an `IEnumerable`, that lazily loads new pages of results on demand. `ToList()`, on the other hand, returns a `List`, that is immediately loaded with all of the results, possibly from many pages.

  These two methods are available on all of the `WikiQuery` types, including `WikiQueryResult` and return the result based on the type parameter `TSource` for most of the types. And exception is `WikiQueryResult`, which uses a separate `TResult` type parameter.

- `OrderBy()` (and `OrderByDescending()`) sets the ordering. Because it doesn't make sense to sort the same query multiple times and because no module supports sorting by multiple keys, this method returns the type with `Sortable` removed.

  This method is available on the two `Sortable` types and uses the type parameter `TOrderBy`.

- `Pages` is a property that returns a `PagesSource` that can then be used in a `prop` query. See Section 4.1.3 for more information.

  This property is available on the two `Generator` types and uses the type parameter `TPage`.

For a state diagram of transitions between the `WikiQuery` types and other related types, see Figure 4.1.

4.1.3 `PagesSource`

The `PagesSource` type represents a collection of pages that can be used in `prop` queries, to get information about those pages. This information can be for example a list of categories for each page in the collection.

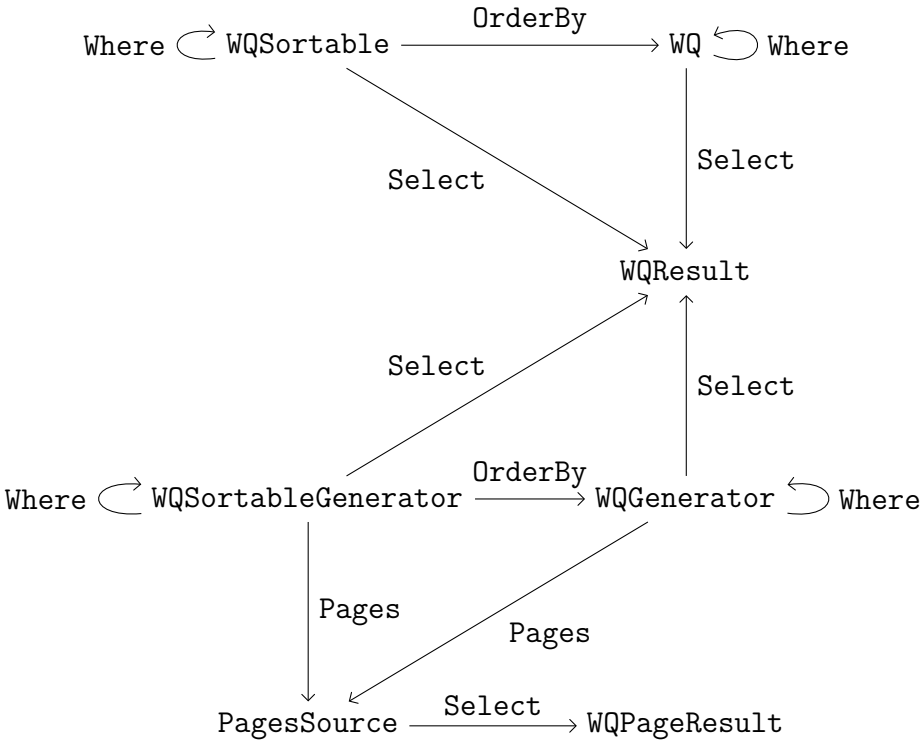

Figure 4.1: State diagram of `WikiQuery`-related types
(`WikiQuery` is shortened to `WQ` to save space)

There are two kinds of `PagesSource`s: generator-based and list-based.

List-based sources use a static list of pages, given as a collection of page titles, page IDs or revision IDs.

Because the number of pages given this way in a single API request is fairly limited (usually to 50), large lists have to be queried multiple times. `PagesSource` handles this transparently, so the user can input as many pages as he wants and doesn't have to worry about the limit.

One exception is if the limit is different than the default of 50 for the current user on the current wiki. In that case, the user should change the limit by setting the static property `ListPagesCollection.MaxLimit`.[1]

If the collection used to create a `PagesSource` is lazy, it is iterated in a lazy manner. For example, it could be the result of another LinqToWiki query, with additional processing by LINQ to objects, that is not possible using LinqToWiki alone. Or it could the result of a query from another wiki. In such cases, the original query will only make as many requests as necessary for the follow-up query.

Generator-based sources represent a dynamic list of pages that is the result of another API query, like the list of all pages on a wiki from the `allpages` module. This way, the list of pages doesn't have to be retrieved separately, only to be sent back.

[1] In all other cases where limits are important in this library, they limit the output, not the input. That is why simply setting `limit=max` works in those other cases, but doesn't work here.

Generator queries also have to handle paging, as described in Section 2.1.1, including the exception for the `revisions` module.

Thanks to the fact that both kinds of page sources for `prop` queries are represented by the same (abstract) type, the user of this library can use the same code to work with any source, thus avoiding repetitive code.

To actually create a `prop` query for a page source, one uses the `Select()` method. Its parameter is a lambda, whose parameter is the type parameter `TPage` of `PagesSource`. This type is the same for all queries on the same wiki, but could be different for diffrent wikis.

Inside the lambda, properties and methods of the `TPage` type can be accessed. Each of them represents a `prop` module and all of the methods return one of the `WikiQuery` types, which can then be queried as usual, with one condition: the `WikiQuery` types can't "leak" outside of the query, so one has to use `ToEnumerable()` or `ToList()` inside the lambda.

There is a special case for the `revisions` module, which can be also used with the `FirstOrDefault()` method, which means only the most recent revision for each page is selected.

If a `prop` module has a single result (not a collection), it is represented as a property that directly returns this result, no querying is possible.

The methods of these `prop` queries are inside a lambda expression, so they are not actually executed unless the expression was compiled and the resulting delegate invoked. Because of this, processing them is not as simple as with normal queries. For more details, see Section 4.1.6.

For an example of `PagesSource` query, see Figure 4.2.

```
pagesSource.Select(
    p =>
    new
    {
        p.Info,
        Categories =
            p.Categories()
            .Where(c => c.Show == Show.NotHidden)
            .Select(c => new { c.Title, c.SortKeyPrefix })
            .ToEnumerable()
            .Take(10)
        }
)
```

Figure 4.2: An example of `PagesSource` query that uses the `info` and `categories` modules

4.1.4 `QueryParameters`

The `QueryParameters` type contains the parameters of a query:

- sort direction and parameter by which to sort,
- list of properties to select and a delegate that uses them to construct the result object,
- list of other parameters, as key-value pairs.

`QueryParameters` is an immutable type, so that one beginning of a query can be safely used repeatedly, as is the case with LINQ to objects. The list of other parameters is a functional-style immutable linked list.

The `PropQueryParameters` type derives from `QueryParameters` and is used to store information about a single module in a `prop` query. Apart from inherited members, it also contains the name of the module and a special value indicating whether to retrieve only the first item, which corresponds to the usage of the `FirstOrDefault()` method.

A related type is `PageQueryParameters`, which represents a whole `prop` query. That means it contains a list of `PropQueryParameters` objects and also information about the source of the query.

4.1.5 `ExpressionParser`

The `ExpressionParser` static class is used to process expression trees from LINQ methods and store the processed query parameters in `QueryParameters`.

Common for all expression tree processing is that closed-over local variables contained in the processed lambda, which are represented as members of a compiler-generated closure class, have to be first replaced by their actual value. This is done using `PartialEvaluator` written by Matt Warren [4].

Also, some property names have to be translated from their C# version to their API version. For details and the reason why this is necessary, see Section 4.2.

Each of the methods requires different processing. Specifically:

- Expression trees from `Where()` are first split into one or more subexpressions that are *and*ed together (`x => subexpr1 && subexpr2 && ...`; *or* is not supported by the API) and each of the subexpressions is then added as a key-value pair to the result.

  Each subexpression has to be in the form `x.Property == Value`, where `Value` is a constant, possibly from an evaluated closed-over variable. The reverse order (`Value == obj.Property`) is also allowed. An alternative for boolean properties is accessing the property directly (`x.Property`) or negated (`!x.Property`).

- Processing `OrderBy()` expression trees is simple: they can either be identities (`x => x`), which means default sorting will be used (which is the only possibility for some modules), or they can be simple property accesses (`x => x.Property`), which means the result will be sorted by that property.

The order of sorting (ascending or descending) is decided by the method used: whether it was `OrderBy()` or `OrderByDescending()`.

- Expression trees from `Select()` are processed in two steps. First, the expression is scanned for usages of its parameter. If any of its properties are used, it means those properties have to be retrieved from the API. If the parameter is used directly, without accessing its properties, it means all of the properties have to be retrieved, because it is impossible to say which of them will be used.

  For example, the expression `x => new { x.Property1, x.Property2 }` means only `Property1` and `Property2` have to be retrieved. On the other hand, `x => SomeMethod(x)` means all of the properties have to be retrieved.

  Second step is compiling the expression into a delegate, which will then be executed for each item coming from the API.

  Put together, these two steps mean that `Select()` can be used with any expression and only properties that are actually needed will be returned by the API.

4.1.6 PageExpressionParser

The class `PageExpressionParser` is used to process the `Select()` lambda in `PagesSource` queries. The difficulty there is that the direct approach of building the query step-by-step, used in normal queries, will not work. That is because the expression has to be analyzed before there is any page object that it expects as its parameter.

The result of this analysis is twofold: the set of parameters needed for all of the `prop` queries, as a collection of `PropQueryParameters`, and a delegate that can be used to get the result object for each page in the API response.

Because the subquery for each `prop` module has to end with a call to `ToEnumerable()` or `ToList()`, the parameters can be extracted by invoking the part of the subquery before that call. At the beginning of each subquery is invoking a module-specific method on the page object. But because there is no page object to use, that invocation is first replaced by an appropriate `WikiQuery` object.

For example, for the query in Figure 4.2, the invoked code is (where `wikiQuery` is the appropriate `WikiQuery` object):

```
wikiQuery.Where(c => c.Show == Show.NotHidden)
         .Select(c => new { c.Title, c.SortKeyPrefix })
```

To get the delegate, all calls to `Where()` and `OrderBy()` are removed, because their only purpose is to modify the query parameters. Then the single parameter of type `TPage` is replaced by a parameter of type `PageData` and calls to module methods are replaced by calls to `GetData()`, with a type parameter specifying the type of the result and a parameter specifying the name of the module.

The `GetData()` method returns a collection, so for modules that return only a single item, like `info`, a call to `SingleOrDefault()` is also added.

For example the expression in the query in Figure 4.2 is transformed into:

```
pageData =>
new
{
    Info = pageData.GetData<InfoResult>("info")
                   .SingleOrDefault(),
    Categories =
        pageData.GetData<CategoriesSelect>("categories")
                .Select(c => new { c.Title, c.SortKeyPrefix })
                .Take(10)
}
```

4.1.7 Other types

The `QueryProcessor` type manages downloading the result and transforming it from XML to objects. For queries whose result is a collection, it also handles returning the pages in a lazy manner and downloading the follow-up pages when necessary.

The `QueryPageProcessor` type does the same for `PagesSource` queries.

The `Downloader` type takes care of forming the query string, executing the request and returning the result as an `XDocument`. `XDocument` is a part of LINQ to XML, a part of .Net framework for manipulating XML documents.

`Downloader` always uses POST and formats its requests as `application/x-www-form-urlencoded`. This means that all modules work, including those that require POST. On the other hand, uploads of files don't work, because they require `multipart/form-data`.

The decision to use `application/x-www-form-urlencoded` follows from the fact that `multipart/form-data` is very inefficient when sending multiple parameters with short values, which is common when making requests to the API.

4.2 The LinqToWiki.Codegen project

The LinqToWiki.Codegen project contains code that retrieves information about API modules in some wiki, then uses that information to generate C# code to access those modules using Roslyn and finally compiles the code into a library.

Roslyn was chosen, because it is superior when compared with common approaches for code generation in .Net, namely Reflection.Emit and CodeDOM.

Reflection.Emit [5] is a set of types that allow code generation of code at runtime. The generated code can then be directly executed or saved as an assembly (.dll or .exe) to disk. The distinguishing feature is that it uses the low-level Common Intermediate Language (CIL), which means writing any code beyond the simplest methods can be very tedious and error-prone.

CodeDOM [6] can be used to generate code and compile it to an assembly. It uses language-independent model, which can be converted to various .Net languages, including C# and VB.NET. This model is also the biggest disadvantage of CodeDOM, because it means it doesn't support all features of C#. For example, even such basic feature as writing a **`static`** class is impossible in the CodeDOM model without using "hacks".

Detailed description of Roslyn is in Section 2.3.

At this point, we have a library (LinqToWiki.Core) that can be used to access the MediaWiki API the way we want from the final generated library. We can also use the same library to get the information we need about the modules of the API from the `paraminfo` module. And we have decided we want to use Roslyn to generate the final library. What remains is to decide what code to generate, how exactly to map the modules, their parameters and their results into the model of LinqToWiki.Core.

There are some decisions that were already made in LinqToWiki.Core (the `sort` and `dir` parameters should map to `OrderBy()`; the `prop` parameter maps to `Select()`), but several other decisions still remain:[2]

- How should the remaining parameters be mapped? Should they all go into `Where()` or somewhere else? Where?
- How should the modules that don't return lists be mapped? LINQ methods are not suitable for them, because they are meant to work with collections.
- How to name the generated types and members? Specifically, how to represent names that can't be used (like those containing special characters) and names that are undesirable (those that conflict with C# keywords). Also, should the generated members follow .Net naming conventions?

Our answers to these questions are in the following couple of sections.

4.2.1 Naming of generated types and members

Let us start with the last question: Should the generated members follow .Net naming conventions? The .Net naming guidelines [7], that are widely followed by

[2] Obviously, both libraries were written alongside each other, to work well together, not one after the other. But we think it's better to describe them this way, separately.

various .Net libraries and the .Net framework itself, state that names of types and public members should use PascalCase, that is, each word of an identifier should start with a capital letter and the identifier should not contain any delimiters (such as underscores).

We would prefer to follow these naming conventions, but, unfortunately, it is not possible. That is because the names of modules, parameters, result properties and almost all enumerated types in the API use names that are all lowercase, without delimiters between words. That means there is no way to figure out which letters in an identifier should be capitalized (apart from the first one).

As one of the more extreme examples, one of the possible values of the `rights` parameter of the `allusers` module on the English Wikipedia is `collectionsaveascommunitypage`. A human can see that the proper name for that value using PascalCase would be `CollectionSaveAsCommunityPage`, but a computer cannot. (Actually, it is possible that the words could be reliably separated using natural language processing, but doing that is outside the scope of this work.)

Because different .Net languages have different sets of reserved identifier names (usually, those are the language keywords) and because libraries written in one language should be usable from other languages, .Net languages provide a way to use their keywords as identifiers. In the case of C#, this is done by prefixing the identifier with an at sign. So, for example, to use `new` as an identifier, one has to write `@new`.

Thanks to this, using keyword-named identifiers is still possible, although slightly less convenient than with normal identifiers. Also, the naming guidelines suggest avoiding keywords as identifiers.

In MediaWiki core API modules, there are four identifiers that are also C# keywords: `namespace`, `new`, `true`, `false`. Out of these, we decided to shorten `namespace` to `ns`, which is a common abbreviation, so the meaning should not be lost. The other three have to be written with `@` (`@new`, `@true` and `@false`) in C#, because we did not find a reasonable alternative for them.

As for special characters, the delimiters hyphen (`-`), slash (`/`) and space appear in some names in the API, but are not allowed in .Net identifiers, so they are replaced by underscores (`_`).

Some names also start with the exclamation mark (`!`), to indicate negation. Such names are translated by prefixing `not_`. So for example, `!minor` (which means that an edit is not a minor edit) is translated into `not_minor`.

One more special case is that some enumerated types allow an empty value. Such value is then represented by the identifier `none`.

Another question is how to name the generated types. There two kinds of generated types: those that represent some enumerated type and those that represent parameters or results of some module.

For the latter kind, it is simple to come up with a convention like naming them by the module name, suffixed by the specific kind of the type (e.g. `blockResult` for the result of the `block` module or `categorymembersWhere` for the type representing `Where()` parameters for the `categorymembers` module).

But for the former kind, the situation is more complicated. Enumerated types do not have names by themselves, they are part of a parameter or property that has a name. The problem is that different modules often have parameters

and properties with the same name, while their type sometimes is the same and sometimes it is not.

So, there are two options: either let the types that look the same actually be the same generated type, or let each parameter and property have its own distinct type. If we merge the types that look the same, we should not use the module name in their name, because one type can be used with different modules. But that means we need to distinguish different types in another way, like a number. But names like `token5` are not very helpful for the user.

Because of that, we chose the other option, which means including the name of the module in the type name. But doing it this way does not eliminate conflicts completely: In the case when a module has a parameter and a property with the name, their types still have to be distinguished. An example of such type name is `recentchangestype2`.

4.2.2 Structure of generated code

At the start of each query is the `Wiki` type. It contains methods for non-query modules as well as methods to create list-based `PagesSource`s. It also contains the property `Query` that returns an object that contains methods for `list` and `meta` query modules. (`prop` query modules work differently, for more information, see Section 4.1.3.)

With modules that don't return lists, the situation is mostly simple: there are no parameters to sort or filter the result (because it's not a list) and most of those modules also don't have parameters to choose the result properties.

Because of that, a method for each such module, that directly returns the result object is enough. This method has parameters corresponding to the parameters of the module, where required parameters of the module are mapped as normal method parameters and parameters that are not required are mapped as optional parameters. The code of this method builds `QueryParameters` from the method parameters and then executes the query using `QueryProcessor`.

On the other hand, list modules can have several kinds of parameters:

- Those that affect order of the items in the list. They are naturally mapped as `OrderBy()`. The parameters `sort` and `dir` belong here.
- Those that choose what properties appear in the result. They are naturally mapped as `Select()`. Only the parameter `prop` belongs here.
- Those that filter what items appear in the result. They are naturally mapped as `Where()`. For example, the parameters `namespace` and `startsortkey` of the `categorymembers` module belong here.
- Various other parameters. They do not naturally map to any LINQ method. For example, the parameter `title` (that decides which category to enumerate) of the `categorymembers` module belongs here.

The first two kinds are not a problem, because it is clear which parameters belong to them. The second two kinds are a problem, because there is no clear way to automatically distinguish between the two. One exception is if a parameter

is required (as indicated in its description), then it means it belongs to the other parameters.

Required parameters are given as parameters of the module methods, but we decided to treat all non-required parameters that do not belong to the first two kinds, as if they were `Where()` parameters. Unfortunately, this means that some queries do not logically make sense, if we consider that the `Where()` method should only filter the results.

For example, consider this query:

```
wiki.Query.categorymembers()
    .Where(cm => cm.title == "Category:Query languages")
```

There, the `title` property does not actually represent filtering by the title of the category member, it decides which category to enumerate. And without it, the query would not even execute successfully (the parameter `title` is not marked as required, because the parameter `pageid` can be used instead of it).

Proper solution to this problem would require human interaction when generating the code, to choose which parameters belong to `Where()` and which do not. As an alternative, the description of each parameter in the `paraminfo` module could contain its kind.

One more question is how to represent enumerated types. The answer is seemingly simple: make them **`enum`**s and for those parameters or properties that can have multiple values, use bit flags. But the largest type that can be used as an underlying type for **`enum`** is **`ulong`**, which has 64 bits. That means this will work only if there is no enumerated type in the API, that has more than 64 values and can have multiple values at the same time. Unfortunately, the English Wikipedia has one: the type of the `rights` parameter of the `allusers` module has 106 values and the parameter can have multiple values at the same time.

Because of that, each enumerated type is represented by immutable class deriving from the common base class `StringValue`, with inaccessible constructor and static field for each possible value. Combination of values can be represented as a collection, like with other types.

4.2.3 `Wiki`

The top-level type that manages all code generation is `Wiki` (not to be confused with the generated `Wiki` type from Section 4.2.2). It manages retrieving information about API modules and generating code for them.

When the code generation is complete, it saves the generated C# files to a temporary directory and compiles them using CodeDOM. CodeDOM is used for the compilation, because its compiler is the full C# compiler and can handle all features of C# (unlike the CodeDOM object model). The Roslyn compiler is not able to compile some useful expressions, such as collection initializers (but the object model of Roslyn is complete).

4.2.4 `ModuleSource`

The `ModuleSource` class is used to retrieve information about modules of the API and transform it from XML to objects, like `Module`, `Parameter` and `Parameter-`

`Type`. This information comes from the `paraminfo` module and is fetched using LinqToWiki.Core.

In fact, this code can be viewed as a sample on how to use LinqToWiki.Core without code generated by LinqToWiki.Codegen. Generated code cannot be used to work with the `paraminfo` module, because it is one of the modules, whose response is complicated and does not fit into the simple type system used by `paraminfo`.

Because the addition of result properties to `paraminfo` was made as a part of this work (see Section 3) and so is quite recent, there is also another option to get this information: `ModuleSource` can accept a "props defaults" file, that contains the necessary information. The file looks the same as `paraminfo` response (in XML format), except it contains only the added information. This file can be created from another wiki that can already provide this information, or it can be written by hand. It can be also useful to work with modules from extensions, that currently don't provide this information.

4.2.5 `ModuleGenerator`

`ModuleGenerator` and related types are the ones that actually generate code for each module using Roslyn. Each type generates code for a certain kind of module, so for example `ModuleGenerator` works with non-query modules, while `QueryModuleGenerator` works with most query modules.

Each generator creates all the code that is necessary for that module. For example, for a `list` query module, this includes generating `Where`, `Select` and possibly `OrderBy` classes, method in the `QueryAction` class (which is returned by the `Query` property of the `Wiki` class) and types for all its enumerated types.

4.3 The linqtowiki-codegen application

4.4 Samples of queries

5. Future work

Conclusion

Bibliography

[1] *MediaWiki API*. MediaWiki.org. https://www.mediawiki.org/wiki/API.

[2] Skeet, Jon. *C# in Depth*. 2nd edition. Stamford: Manning, 2011. Part 3, C# 3: Revolutionizing how we code. ISBN 978-1-935182-47-4.

[3] *Microsoft® "Roslyn" CTP*. MSDN. http://msdn.microsoft.com/en-US/roslyn.

[4] Warren, Matt. *LINQ: Building an IQueryable Provider – Part III*. The Wayward WebLog. http://blogs.msdn.com/b/mattwar/archive/2007/08/01/linq-building-an-iqueryable-provider-part-iii.aspx, 2 August 2007.

[5] *Emitting Dynamic Methods and Assemblies*. MSDN Library. http://msdn.microsoft.com/en-us/library/8ffc3x75.

[6] *Dynamic Source Code Generation and Compilation*. MSDN Library. http://msdn.microsoft.com/en-us/library/650ax5cx.

[7] *Guidelines for Names*. Design Guidelines for Developing Class Libraries, MSDN Library. http://msdn.microsoft.com/en-us/library/ms229002.

List of Tables

List of Abbreviations

Attachments