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 **2**

1 Problem analysis **3**

2 Background **5**
2.1 MediaWiki API 5
2.1.1 Paging 6
2.1.2 The `paraminfo` module 9
2.2 LINQ and expression trees 9
2.3 Roslyn 13

3 MediaWiki improvements **15**

4 The LinqToWiki library **17**
4.1 The LinqToWiki.Core project 17
4.1.1 `QueryTypeProperties` 18
4.1.2 `WikiQuery` 18
4.1.3 `Downloader` 19
4.2 The LinqToWiki.Codegen project 19
4.3 The linqtowiki-codegen application 19
4.4 Samples of queries 19

5 Future work **20**

Conclusion **21**

Bibliography **22**

List of Tables **23**

List of Abbreviations **24**

Attachments **25**

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 set 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.

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 are available and that all properties 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. It can be used to remotely access any MediaWiki wiki (unless the API is disabled in the configuration) using the HTTP protocol.

Parameters for the 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 `boolean`s have 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, error alone is returned.

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 a user (it can be used only by a user 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 the categories of all 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 the 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 page.

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 revision 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.

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. 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.

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

```
<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

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 a so 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 function from functional programming. See Figure 2.4 for comparison of some of the LINQ query operators, query expression clauses, and higher-order functions.

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

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

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.

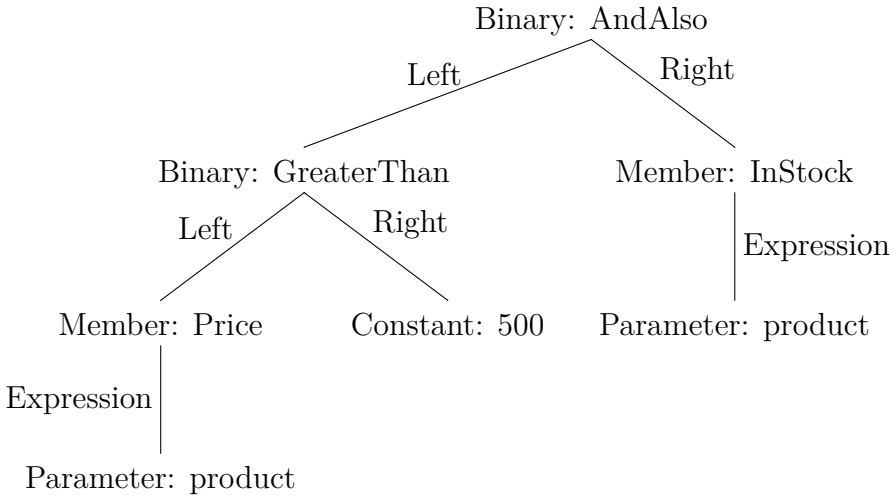

Figure 2.5: The body of the expression tree for the lambda expression `product => product.Price > 500 && product.InStock`

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). 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 its latest public version is a CTP (Community Technology Preview) from October 2011. In the 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 in the CTP, 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; }
```

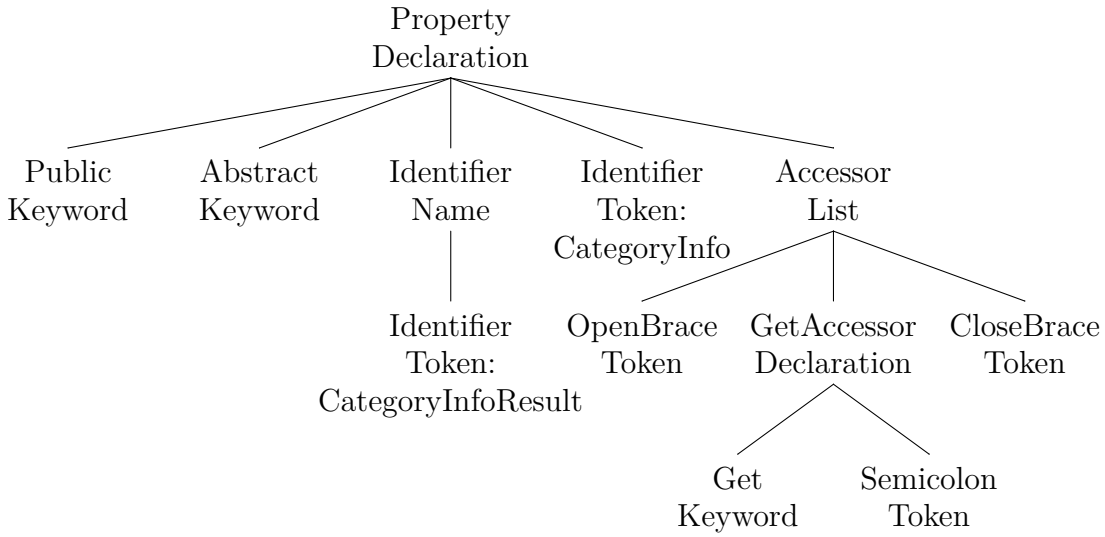

Figure 2.6: An example of a piece of C# code and its Roslyn syntax tree (trivia not shown)

Roslyn syntax trees are immutable and they 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.

For an example of code to manually create the syntax tree from Figure 2.6, see Figure 2.7.

```
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

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 is also 17 modules that can be partially represented using this type system, but not completely.

As of 10 May 2012, the patch that adds this ability to the `paraminfo` module and the necessary information to most other modules is waiting for review by MediaWiki developers.

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 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.

```
<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 using Roslyn. This project also contains helper types for easier creating of Roslyn syntax trees.

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

The LinqToWiki.ManuallyGenerated project is a sample of how one could write code to access a wiki using LiqToWiki 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 application 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. 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. Those information include 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 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.

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

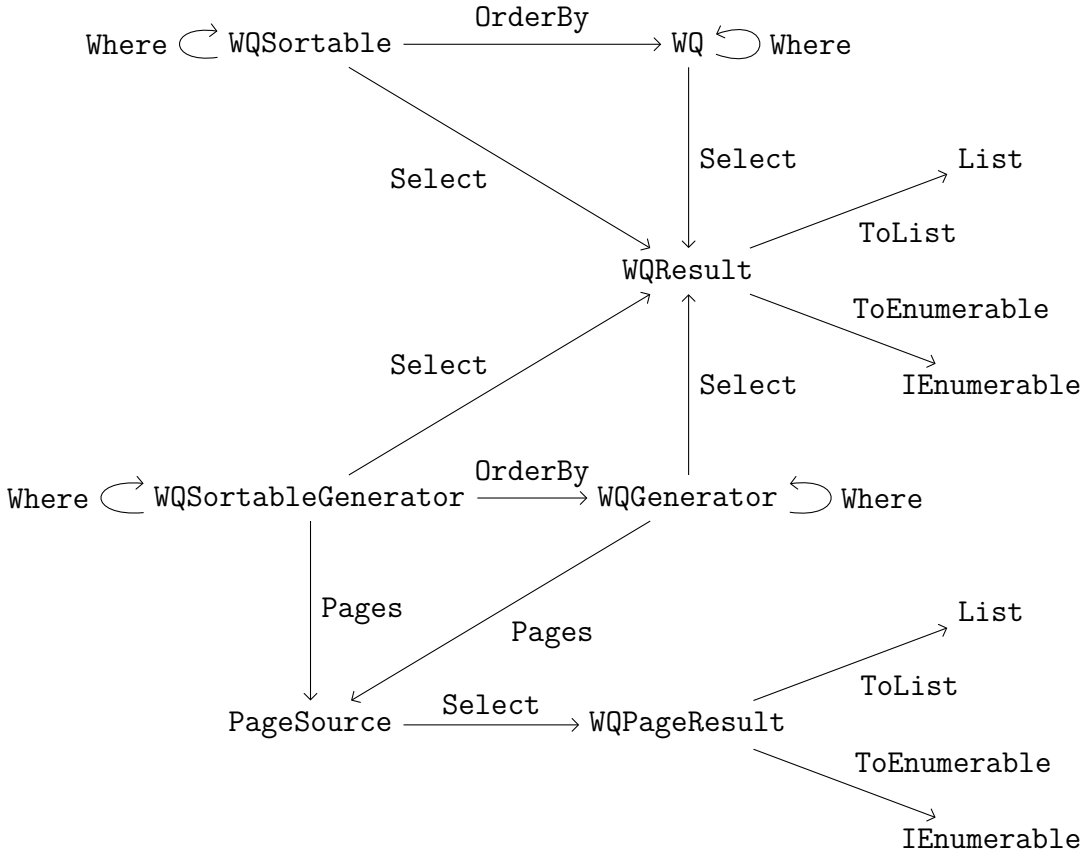

Figure 4.1: State diagram of `WikiQuery`-related types
(`WikiQuery` is shortened to `WQ` to save space)

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 `query.OrderBy(x => x.Title)` would compile and execute fine, but `query.Orderby(x => x.Name)` would fail to compile.

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

4.1.3 `Downloader`

Probably the most basic type in this project is `Downloader`. It takes care of forming the query string, executing the request and returning the result as an `XDocument`.

`XDocument` is 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

4.3 The linqtowiki-codegen application

4.4 Samples of queries

5. Future work

Conclusion

Bibliography

List of Tables

List of Abbreviations

Attachments