LINQ to Wiki

- library for accessing MediaWiki API from .Net
- uses LINQ for querying lists
- strongly-typed (no magic strings)
- the API is big, changes often and can be different for different wikis (thanks to extensions)
  - because of that, Roslyn is used to generate code based on description the API provides about itself

Modules in the API

- the API is divided into modules
- each module has a specific function
    - examples: edit page, list all categories, show categories of a given page
- there are three kinds of modules: simple modules, list modules and prop modules
    - simple modules return a single result
    - list modules return a list of items as a result
    - prop modules are used to retrieve additional information about a list of pages
        - that list can be from a list module or a hard-coded set of pages

Simple modules

- simple modules are represented as methods on the `Wiki` class
- parameters of modules are parameters of the method
  - optional parameters as C# 4 optional parameters
  - most parameters are optional
- the result of the module is the result of the method

```
var wiki = new Wiki("en.wikipedia.org");

// get edit token, necessary to edit pages
var token = wiki.tokens(new[] { tokenstype.edit }).edittoken;

// create new section "Hello" on the page "Wikipedia:Sandbox"
wiki.edit(
    token: token, title: "Wikipedia:Sandbox", section: "new",
    sectiontitle: "Hello", text: "Hello world!");
```

List modules

- list modules are methods on the `Query` property of `Wiki`
- query can be modified by using LINQ methods `Where()`, `OrderBy()` (where available) and `Select()`
  - different properties can be used for each method
    - example: pages can't be ordered or filtered by their text, but the text can be selected
  - query syntax (`from`, `where`, `orderby` and `select`) can be used instead of method syntax
  - other LINQ methods (or query operators) can't be used (won't compile)
  - lambdas are compiled as expressions, parsed into API parameters
  - `Select()` lambda can contain any code, `Where()` and `OrderBy()` can't
- each query ends by a call to `ToEnumerable()` or `ToList()`

List modules example

```
var pages = (from cm in wiki.Query.categorymembers()
             where cm.title == "Category:Query languages"
             orderby cm.sortkey descending
             select cm.title)
            .ToEnumerable();
```

- pages is a lazy list of page titles (`IEnumerable<string>`) in the category “Query languages”, sorted by their “sortkey” in reverse
  - the laziness means that for example enumerating `pages.Take(10)` will only make one request, for the first page of results

Page sources

- prop modules work on page sources (`PagesSource`)
- page source can be created from a static list of pages:

```
var source = wiki.CreateTitlesSource("C Sharp", "LINQ");
```

- or they can be created from some queries without a `Select()` by accessing the `Pages` property:

```
var source = (from cm in wiki.Query.categorymembers()
              where cm.title == "Category:Query languages"
              select cm)
             .Pages;
```

  - `select cm` here doesn't result in a `Select()` call

Prop modules

- given a page source, you can use `Select()` to access one or more prop modules
- the lambda parameter has a member for each prop module
  - prop modules that return lists are methods, whose result can be used with LINQ methods
  - `ToEnumerable()` (or `ToList()`) has to be called at the end of each subquery
  - prop modules that return single result are properties
- the lambda can contain arbitrary code, except for the part of each subquery before `ToEnumerable()`
- it doesn't matter whether the page source is from another query or from a static list

Prop modules example

```
var pages = source
    .Select(
        p =>
        new
        {
            p.info,
            categories =
            p.categories()
                .Where(c => c.show == categoriesshow.not_hidden)
                .Select(c => new { c.title, c.sortkeyprefix })
                .ToEnumerable()
                .Take(1)
        }
    ).ToEnumerable();
```

- `pages` is a lazy collection of anonymous objects, each containing the basic information about a page, along with `title` and `sortkeyprefix` of the first category of that page that is not hidden

Code generation

- the `linqtowiki-codegen` console application can be used to generate assembly for a given wiki
- it should be run again when the API changes or for working with different wiki
- but if the changes are not relevant, the generated assembly for an older version or for different wiki can be used
- the code can be generated into a specified namespace, so that assemblies for different wikis could be used from one application
- the application requires Roslyn

LINQ implementation

- the desired behavior of LINQ methods is achieved by a group of types whose names start with `WikiQuery`
- each type defines the members it can support
- invoking a member stores the changes it made and can return a different type

$$\begin{tikzcd}
\texttt{WikiQuerySortable} \arrow[loop left, "\texttt{Where}"] \arrow[r, "\texttt{OrderBy}"] \arrow[dr, "\texttt{Select}"'] & \texttt{WikiQuery} \arrow[loop right, "\texttt{Where}"] \arrow[d, "\texttt{Select}"] \\
 & \texttt{WikiQueryResult} \\
\texttt{WikiQuerySortableGenerator} \arrow[loop left, "\texttt{Where}"] \arrow[r, "\texttt{OrderBy}"] \arrow[ur, "\texttt{Select}"] \arrow[d, "\texttt{Pages}"] & \texttt{WikiQueryGenerator} \arrow[loop right, "\texttt{Where}"] \arrow[u, "\texttt{Select}"'] \arrow[dl, "\texttt{Pages}"] \\
\texttt{PagesSource} \arrow[r, "\texttt{Select}"] & \texttt{WikiQueryPageResult}
\end{tikzcd}$$

Paging implementation

- lists can have thousands or even millions results, so the API returns them in pages
- each pages is usually 500 items (the limit is raised to 5000 for some users)
- queries using prop modules have two kinds of paging: for the source list and for the results
- LINQ to Wiki handles paging transparently for users
  - in the case of static page source, the list is split into pages by LINQ to Wiki
  - otherwise, it is handled by the API and LINQ to Wiki only has to pass the correct paging parameters to the query
- `ToEnumerable()` returns a lazy collection, which means only the necessary pages are retrieved
- in prop modules queries, both kinds of paging can be lazy independently

Codegen implementation

- information about modules necessary to generate the code is returned by the `paraminfo` module
  - LINQ to Wiki is used internally to access this module, although with the part that is usually generated written manually
- the module returns description of parameters and result properties of modules
  - the types of parameters or properties can be either simple types (e.g. string, integer) or enumerated types
  - some enumerated types can be combined together and they can also have more than 64 values, so they are not generated as `enum`s (because they couldn't be flag enums)
  - some properties have more complex types, these are not represented in `paraminfo`, so they aren't present in LINQ to Wiki either
- Roslyn is used to generate the code through a helper class that makes the code simpler
- CodeDOM is used to actually compile the generated code, because Roslyn compiler does not support all features of C# yet (like object initializers)

Codegen implementation

- the following types are always generated: `Wiki` for simple modules, `Query` for list modules and `Page` for prop modules
- for each simple module, its result type is also generated
- for each list modules, `Where`, `Select` and possibly `OrderBy` type is generated
- prop modules behave as simple modules or list modules in this regard, depending on whether they return single item or a list
- a type is also generated for each enumerated type that is used by a parameter or a property