

# FRED API Toolkit

## Overview

This toolkit provides a simple and easy-to-use way to consume economic data from the FRED API. The toolkit removes the complexities of accessing the FRED API and gives users a timesaving and straightforward alternative to building the access infrastructure themselves.

The toolkit is fully extensible allowing users to add functionality as necessary. It is written in the .Net C# language but is compliant with other .Net languages and can be easily combined with them as a full solution.

In designing this toolkit, our primary focus was the user, and based on that focus, these goals naturally followed:

- Give users an easy and intuitive way to consume the FRED API based on knowledge they already have.
- Use vocabulary that is part of the FRED API. Users need only learn the FRED API vocabulary because the same vocabulary extends to this toolkit.
- Show users only choices that apply to the task they are doing so that they don't have to be concerned about which choices apply and which do not.
- Allow users to customize behavior but don't force them to.
- Keep the amount of code necessary to get the job done to a minimum.
- Use a design that moves more complex code away from the user by providing *façades*<sup>i</sup>. The term “façade” is used throughout the project and this document.

Meeting these goals provides software with a low barrier to entry and a flat learning curve that allows users to write and test consumer code quickly and easily. We believe this has a fundamental and direct positive impact on the productivity of the teams and organizations that use this toolkit.

## Components

Using the FRED API as a contextual starting point, it's helpful to view each API operation as consisting of three components within the toolkit:

1. The argument values that are inputs to each API request;
2. The output values from the API response;
3. The façade that serves as the transmission mechanism for arguments and results between user code and the Fred API.

Here is an example of these components for the **Categories** [fred/category](#) API operation:

### Categories

- [fred/category](#) - Get a category.
- [fred/category/children](#) - Get the child categories for a specified parent category.
- [fred/category/related](#) - Get the related categories for a category.

- The façade is a [Category](#) object.
- The arguments are contained in a [CategoryArguments](#) object.
- The response values from FRED are contained in a [CategoryContainer](#) object.

Notice how these object class names follow the FRED naming conventions as closely as possible. There are further examples throughout this document of how the toolkit uses names that are already familiar to users who are accustomed to the FRED API.

Each toolkit class related to an operation within FRED's **Categories** API is a member of the [FRED.API.Categories](#) namespace. A parallel exists for each of the other API groups: **Releases**, **Series**, **Sources** and **Tags**. Each class that a user typically needs to reference in code is uniquely named. This means that namespaces are not necessary to resolve ambiguities.

## Deciding Which Classes to Use

In deciding which toolkit classes to use, the user needs to answer several questions:

1. What FRED API operation am I interested in?
2. What type of response do I want (XML, JSON or .Net object)<sup>1</sup>
3. What arguments do I need for the operation and what are their values?
4. Do I need to invoke the operation asynchronously?

The answers to these questions will define the classes the user employs in code. Here's an example.

1. I'm interested in the [fred/category/children](#) operation.
  - API façade: [CategoryChildren](#)
  - Response: unknown at this point
2. I want the response to be a .Net object

---

<sup>1</sup> The [fred/series/observations](#) API supports two additional response types, tab-delimited text and Excel, which are addressed separately below.

- API façade: `CategoryChildren` (no change from above)
  - Response: `CategoryContainer`
3. I want to specify only category id and let all others default
- API façade: `CategoryChildren` (no change from above)
  - Response: `CategoryContainer` (no change from above)
  - The argument name will be `category_id`, the same as in the FRED API. The façade object offers the arguments as a property so there is no need to create an arguments object. If the user wanted to create a new arguments object, its class would be `CategoryChildrenArguments` for this example.
4. I don't need an asynchronous invocation. Synchronous execution is fine.
- No change to any class.

Given the answers in the example above, here is the code that's required:

```
CategoryChildren api = new CategoryChildren { ApiKey = "your key" };
api.Arguments.category_id = 13;
CategoryContainer container = api.Fetch();
```

Changing the choices the user makes in answering the questions requires only minor changes to the code. Answering question #2 differently results in the following C# code:

2. I want the response to be XML
- API façade: `CategoryChildrenXml`
  - Response: `string`

```
CategoryChildrenXml api = new CategoryChildrenXml { ApiKey = "your key" };
api.Arguments.category_id = 13;
string container = api.Fetch();
```

Everything else remains unchanged. Similarly, to request a JSON response, the code is:

```
CategoryChildrenJson api = new CategoryChildrenJson { ApiKey = "your key" };
api.Arguments.category_id = 13;
string container = api.Fetch();
```

One thing that should stand out among these examples is that the method invoked on the façade object is the same: `Fetch()`. There is no reason to complicate the code by using names

specific to an API operation; the façade and arguments class names already separate FRED operations from one another, and the behavior across all operations is the same, namely, fetch the data.

The single divergence from this protocol is for asynchronous invocations. Instead of `Fetch()`, the method becomes `FetchAsync()`. Responses continue to use the class naming conventions already discussed, but each class defines a `Task` and the `await` keyword is added to define a continuation point for the asynchronous operation.

```
CategoryChildren api = new CategoryChildren { ApiKey = "your key" };
api.Arguments.category_id = 13;
Task<CategoryContainer> task = api.FetchAsync();
CategoryContainer container = await task;
```

Or more simply (with no additional code needed prior to the continuation point):

```
CategoryChildren api = new CategoryChildren { ApiKey = "your key" };
api.Arguments.category_id = 13;
CategoryContainer container = await api.FetchAsync();
```

Note that the `async` keyword (not shown) is added to the method that contains the asynchronous invocation. This is a requirement of .Net, not of the toolkit.

Below is a summary of choices for the [fred/category/children](#) API operation.

<u>Invocation</u>	<u>Response</u>	<u>Façade class</u>	<u>Response class</u>	<u>Method</u>
Synchronous	.Net object	CategoryChildren	CategoryContainer	Fetch()
	XML	CategoryChildrenXml	string	Fetch()
	JSON	CategoryChildrenJson	string	Fetch()
Asynchronous	.Net object	CategoryChildren	CategoryContainer	FetchAsync()
	XML	CategoryChildrenXml	string	FetchAsync()
	JSON	CategoryChildrenJson	string	FetchAsync()

Some noteworthy things about this summary are:

1. The façade class describes both the FRED API endpoint and the response format.
2. There are only two choices for response class, based on whether the response is a .Net object or a string-oriented response.
3. There are only two choices for method, based on whether the invocation is synchronous or asynchronous.

This succinct protocol, along with a naming convention that closely parallels that of FRED, provides an intuitive foundation for writing code to consume the FRED API through the toolkit.

In addition to the three response types mentioned above, the FRED API also supports two additional response options for the [fred/series/observations](#) API endpoint: saving tab-delimited text and Excel files, in zipped format, to a file system location. The decision process is similar to that used for the other three response types.

Below is an extended summary of choices for the [fred/series/observations](#) API operation.

<u>Invocation</u>	<u>Response</u>	<u>Façade class</u>	<u>Method</u>
Synchronous	Text	<a href="#">SeriesObservationsText</a>	Fetch(filename, true)
	Excel	<a href="#">SeriesObservationsExcel</a>	Fetch(filename, false)
Asynchronous	Text	<a href="#">SeriesObservationsText</a>	FetchAsync(filename, true)
	Excel	<a href="#">SeriesObservationsExcel</a>	FetchAsync(filename, false)

Notice that the Façade classes follow the same general naming protocol as discussed previously, using the “[Text](#)” and “[Excel](#)” extensions to indicate the desired file format. The response (not shown) for each of the four choices is a Boolean, indicating whether or not the operation succeeded. The methods are the familiar Fetch() and FetchAsync(), each taking a string containing the name of a file system location where the response is stored, and an indicator as to whether the contents of an existing file can be overwritten; the default for overwriting is true and can be omitted.

## Arguments

As mentioned in the **Components** section above, argument classes follow the names of their respective FRED API operation, with “[Arguments](#)” appended to the operation name. Consider this subset of operations:

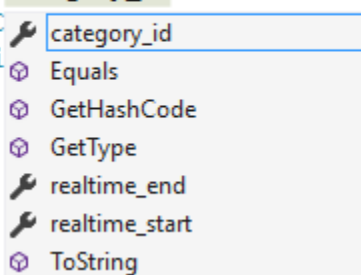
### Categories

- [fred/category](#) - Get a category.
- [fred/category/children](#) - Get the child categories for a specified parent category.
- [fred/category/related](#) - Get the related categories for a category.

The corresponding argument classes are [CategoryArguments](#), [CategoryChildrenArguments](#), and [CategoryRelatedArguments](#).

Each arguments class offers a set of properties related to the context in which the class is used. For example, the context for the [CategoryChildrenArguments](#) class is the [fred/category/children](#) API operation. The [CategoryChildren](#) API façade is the only façade that exposes this particular arguments class. Other API façades expose the arguments class appropriate to its (the façade’s) API context.

```
CategoryChildren api = new CategoryChildren { ApiKey = "your key" };
api.Arguments.category_id = 13;
Task<CategoryC
CategoryContai
```



```
category_id int CategoryChildrenArguments.category_id { get; set; }
```

The code above shows Visual Studio® 2015’s choices for the [CategoryChildrenArguments](#) instance. The API façade exposes this instance through its Arguments property. All choices in the arguments class apply, meaning the user doesn’t have to decide which are appropriate for their context and which are not. The three available arguments for this operation are `category_id`, `realtime_end`, and `realtime_start`. The four methods shown in the list are common to all .Net instances and are not part of the toolkit.

The code above offers a convenient method of setting argument values, since the API façade exposes an arguments instance. Alternatively, the user can instantiate the appropriate arguments class, set the instance's properties, and set the façade's Arguments property to that instance. The code below shows an example of this.

```
CategoryChildren api = new CategoryChildren { ApiKey = "your key" };
CategoryChildrenArguments arguments = new CategoryChildrenArguments
{
    category_id = 13
};
api.Arguments = arguments;
CategoryContainer container = api.Fetch();
```

This might be appropriate if the arguments instance is obtained from another source. Otherwise, the shorthand method is more concise.

## Argument Validation

The value for some arguments is constrained by an enumerated type, so the argument value is guaranteed to be valid.

```
CategorySeries api = new CategorySeries { ApiKey = apiKey };
api.Arguments.category_id = 125;
api.Arguments.sort_order = FREDData.sort_order_values.|
```

FREDData.sort\_order\_values.asc = 0

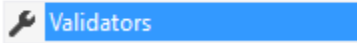
Other arguments are unconstrained, like a string that can contain any value. The remaining category is that which is constrained with values too broad to fit conveniently in an enumeration.

An example is the limit property that is part of many arguments classes. The value for this property is generally constrained between 1 and 1000, according to the FRED API web site. While a user interface might apply a constraint through a dropdown or other UI widget, etc., the toolkit is user interface-neutral, so the property must be validated.

The user has the choice of validating argument values through the toolkit prior to a call to the underlying FRED API operation or letting the FRED API validate and return a message for any invalid argument. The default is to validate through the toolkit, which is more efficient since it eliminates unnecessary API calls.

Each API façade's Arguments property exposes a Validators property.

```
CategorySeries api = new CategorySeries { ApiKey = apiKey };  
api.Arguments.validator
```



This property holds an instance of the `Validators` class, which itself is a list of `Validator` instances. The `Validators` instance is responsible for iterating through its list of `Validator` instances. The `Validators` container gives each `Validator` the chance to validate a given Arguments property. If a `Validator` is configured specifically for a particular Arguments property, it will validate the property; otherwise it will allow succeeding `Validator` instances the chance to validate the property.

This behavior is an implementation of the Chain of Responsibility design pattern<sup>ii</sup>. The benefit of this design is that the user can accept the default validation rules or configure these rules using custom `Validator` classes. It also provides easy extensibility for future changes to the FRED API.

The user can also opt out of toolkit validation and allow the FRED API to validate all Arguments values. To do this, the code simply clears the `Validators` instance.

```
CategorySeries api = new CategorySeries { ApiKey = apiKey };  
api.Arguments.Validators.Clear();|
```

Setting the instance to null will not have the same effect, as the toolkit instantiates a new `Validators` instance containing default `Validator` instances if a `Validators` instance does not exist.

When toolkit validation is used, any invalid Arguments value causes the toolkit not to call the FRED API with these values. Instead, the toolkit returns a null value for the `Fetch()` response and provides a set of messages to the user detailing the problem(s).



```
CategorySeries api = new CategorySeries { ApiKey = apiKey };
api.Arguments.category_id = 125;
api.Arguments.offset = 0;
SeriesContainer container = api.Fetch();
```

container null

Each API façade's Arguments property exposes a ValidationErrors property. When all arguments validate, this property is null; otherwise its value is a [Dictionary](#) containing, for each validation error, the Arguments property name as key and the validation result as the corresponding value.

```
CategorySeries api = new CategorySeries { ApiKey = apiKey };
api.Arguments.valid
```

ValidationErrors  
Validators

Appendix A details the default property validation chain.

## Calling the FRED API

The Fetch() and FetchAsync() methods contain the behavior to execute a web request to consume the FRED API endpoints. If the request is successful, the toolkit simply returns the results in the requested format as already discussed. But what happens if an exception occurs? The toolkit does not throw exceptions based on errors it receives from the FRED API. Instead it returns null as the response and passes any message from the FRED API back to the client code in the API façade's FetchMessage property. Client code should check the response object for null and optionally process the message that the toolkit exposes.

```

CategorySeries api = new CategorySeries { ApiKey = apiKey };
api.Arguments.category_id = 125;
api.Arguments.tag_names = "sfa";
SeriesContainer container = api.Fetch();
if (container == null)
{
    string message = api.FetchMessage;

```



message 🔍 "Bad Request. Value \"sfa\" for variable tag\_names does not exist." ⇐

Notice in the example that toolkit validation is still enabled by default. The tag\_names property cannot be validated prior to calling the FRED API without the toolkit knowing all allowed values for the Arguments property. Currently, the toolkit does not support this level of validation.

The “Bad Request” response from the FRED API causes the .Net framework to create and throw a .Net exception. The toolkit handles these exceptions without allowing them to propagate back to the client.

Other exceptions will also not propagate back to the client code, but instead appear in the API façade’s Exception property. This design provides a consistent exception handling scheme between synchronous and asynchronous calls.

```

CategorySeries api = new CategorySeries { ApiKey = apiKey };
api.Arguments.category_id = 125;
SeriesContainer container = api.Fetch();
if (api.Exception == null)
{
    ▶ 🔍 api.Exception null ⇐
}

```

The API façades also expose a URL property, shown below. This property contains the actual URL used in the web request to the FRED API.

```
CategorySeries api = new CategorySeries { ApiKey = apiKey };  
api.Arguments.category_id = 125;  
SeriesContainer container = api.Fetch();  
string url = api.ur
```



URL

One API with extended behavior, as discussed above, is the [fred/series/observations](#) API. This API provides behavior for saving zipped files (compressed folders) to a file system location. Each zipped file contains one or more individual files in either tab-delimited or Excel format. Below is an example of code that uses these API extensions.

```
SeriesObservationsText api = new SeriesObservationsText { ApiKey = apiKey };  
api.Arguments.series_id = "GNPCA";  
bool success = api.Fetch(@"C:\Users\SomeUser\Text.zip");
```

This example shows a request for a zipped, tab-delimited text file to be saved on the user's file system. The request is synchronous. The second argument in the Fetch() is omitted, indicating that if the file exists, the operation will overwrite its contents; this is the default. Below is a similar example showing an asynchronous request, Excel format, and no file overwriting.

```
SeriesObservationsExcel api = new SeriesObservationsExcel { ApiKey = apiKey };  
api.Arguments.series_id = "GNPCA";  
bool success = await api.FetchAsync(@"C:\Users\SomeUser\Text.zip", false);
```

When no overwriting is specified and the file already exists, the toolkit does not invoke the FRED API. Instead, the request returns false as the response and the API façade exposes an [Exception](#). Here is an example:

```
SeriesObservationsExcel api = new SeriesObservationsExcel { ApiKey = apiKey };
api.Arguments.series_id = "GNPCA";
string filename = @"C:\Users\Guest\SeriesObservations.zip";
bool success = api.Fetch(filename, false);
```

success false

api.Exception {"File C:\Users\Guest\SeriesObservations.zip already exists."}

The result is similar for other file system exceptions. In these cases, the toolkit invokes the FRED API and returns false and the resulting [Exception](#) in the API façade's Exception property:

```
SeriesObservationsExcel api = new SeriesObservationsExcel { ApiKey = apiKey };
api.Arguments.series_id = "GNPCA";
string filename = @"C:\Users\Who\SeriesObservations.zip";
bool success = api.Fetch(filename, false);
```

success false

api.Exception {"An exception occurred during a WebClient request."}

api.Exception.InnerException {"Could not find a part of the path 'C:\Users\Who\SeriesObservations.zip'."}

Here is an example of an API call that includes code summarizing many of the topics covered.

```
CategoryRelatedTags api = new CategoryRelatedTags { ApiKey = apiKey };
api.Arguments.category_id = 125;
api.Arguments.tag_names = "services;quarterly";

Task<TagContainer> task = api.FetchAsync();
// do some work here independent of the fetch results
TagContainer container = await task;

// for synchronous behavior, replace the three lines above with the following
// TagContainer container = api.Fetch();

if (container == null)
{
    // this logic could be factored out for reusability
    if (api.Exception != null)
    {
        // process the exception
    }
    if (api.FetchMessage != null)
    {
        // process the response message
    }
    if (api.Arguments.ValidationErrors != null)
    {
        // process the validation errors
    }
}
else
{
    // process the TagContainer response
}
```

## Appendix A – Property Validation Chain

The following table shows, from top to bottom, the toolkit’s default chain of responsibility for property validation. When a property is validated and considered valid, validation for that property is complete; the remainder of the chain is not considered.

Property Name(s)	Validation	Applies To
release_id	Required and positive.	Release ReleaseDates ReleaseSeries ReleaseSources ReleaseTags ReleaseRelatedTags
series_id	Required.	Series SeriesCategories SeriesObservations SeriesRelease SeriesTags SeriesVintageDates
source_id	Required and positive.	Source SourceReleases
tag_names	Required.	CategoryRelatedTags ReleaseRelatedTags SeriesSearchRelatedTags RelatedTags TagsSeries
search_text	Required.	SeriesSearch
series_search_text	Required.	SeriesSearchTags SeriesSearchRelatedTags
limit	Inclusively between 1 and 10,000.	ReleaseDates SeriesVintageDates
	Inclusively between 1 and 100,000.	SeriesObservations
	Inclusively between 1 and 1000.	All others
offset	Positive or zero.	All
tag_names and exclude_tag_names	When exclude_tag_names is valued, tag_names must also be valued.	All

The order of the chaining for a particular property is important. In the table, notice that the limit property is validated differently for certain façade classes. This implies that those façade-specific validations must occur before the more general validations, or they may be ignored.

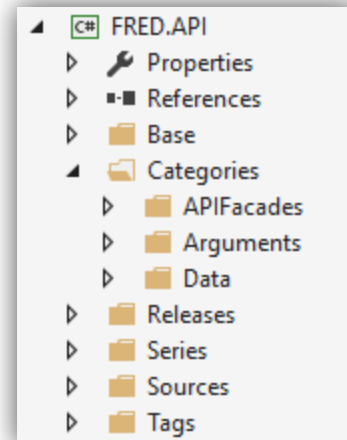
## Appendix B – Namespaces

Namespaces follow a standard naming protocol that reflects the FRED API context of a particular API facade, arguments or data container class. Each namespace begins with the name of the .Net assembly, FRED.API. This is followed by the FRED API group: Categories, Releases, Series, Sources and Tags. After the FRED grouping, the final namespace node is the toolkit group into which the class fits.

To the right is an image of the class hierarchy. Folder names in the hierarchy correspond to namespaces. The Categories node is the only one shown expanded, but the other four have parallel folders. Expanding the APIFacade node would show all the category-related APIFacade classes. So, based on this structure, the fully-qualified name of the [Category](#) API façade class is:

[FRED.API.Categories.APIFacades.Category](#)

All other classes in the five FRED API sections follow the same namespace naming convention.





## Appendix C – Document Conventions

These are the typeface and style conventions used throughout this document.

- The typeface and font size used in the body of the document is Calibri 12 point.
- Section names are shown in Cambria 14 point bold.
- References to FRED API web pages are shown in Cambria 12 point for section names and Verdana 10.5 for active links, for example, **Categories:** [fred/category](#). This is similar to the typeface used on the actual web pages.
- Class names and namespaces are shown in Consolas 11 point, light blue text, for example, `SeriesObservation`.
- Other code is shown in Lucida Sans Unicode 10, for example, `category_id`.

## Notes

---

<sup>i</sup> Design Patterns, Elements of Reusable Object-Oriented Software. Gamma, et al.

Façade is a Structural Pattern. Its description is “Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.”

<sup>ii</sup> Design Patterns, Elements of Reusable Object-Oriented Software. Gamma, et al.

Chain of Responsibility is a Behavior Pattern. Its description is “Avoid coupling the sender of a request to its receiver by giving more than one object the chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.”