URL Query String Building

Using a Fluent approach to build URL query strings

Jonathan Markland

Logo

Description automatically generated

A picture containing icon

Description automatically generated

# Background

I have noticed a common need to build URL strings, including query parameters.

This seem to be done in an ad-hoc manner every time, with different competing implementations in evidence in the code base of the service I looked at.

The process of building a query string is complicated by two factors:

* Deciding which separator to append: (“**?**” or “&”).
* The need to support *conditionally appending* parameters

This document looks at some examples I’ve found on my travels that are the motivation for a library feature.

I present a library solution that aims to satisfy **common case** needs. This drastically shortens the code and eliminate the potential for bugs. I present a comparison of old versus new.

I link to a repository containing standalone code for the proposal. This should be seen as a starting point for discussion, rather than a drop-in feature for the company’s framework.

# Motivating Examples Found

## Example 1

*This is an example I discovered in the existing code base.*

Technique: Repeatedly assign a string reference to decide the separator. Manually concatenate result.

var queryParamSeparator = "?";

if (businessId != null)

{

requestUrl += $"{queryParamSeparator}businessId={businessId}";

queryParamSeparator = "&";

}

if (productLicenseId != null)

{

requestUrl += $"{queryParamSeparator}productLicenseId={productLicenseId}";

queryParamSeparator = "&";

}

if (productType != null)

{

requestUrl += $"{queryParamSeparator}productType={productType}";

queryParamSeparator = "&";

}

if (page != null)

{

requestUrl += $"{queryParamSeparator}page={page}";

queryParamSeparator = "&";

}

if (itemsPerPage != null)

{

requestUrl += $"{queryParamSeparator}page-items={itemsPerPage}";

queryParamSeparator = "&";

}

if (includeSoftDeleted)

{

requestUrl += $"{queryParamSeparator}includeSoftDeleted=true";

}

## Example 2

*This is an example I discovered in the existing code base.*

Technique: Use the **RequestHelper** class.

This still feels very manual, with repeated **null** checks needed for **optional** parameters.

var requestUrl = $"/v1/users/accounts";

if (userId.HasValue)

{

requestUrl = RequestHelper.AddQueryParameter(requestUrl, $"userId={userId.Value}");

}

else if (!string.IsNullOrEmpty(userEmail))

{

requestUrl = RequestHelper.AddQueryParameter(requestUrl, $"email={userEmail}");

}

var requestHeaders = RequestHelper.BuildDefaultRequestHeaders();

if (platform != null)

{

requestUrl = RequestHelper.AddQueryParameter(requestUrl, $"platform={platform}");

}

if (page != null)

{

requestUrl = RequestHelper.AddQueryParameter(requestUrl, $"page={page}");

}

if (itemsPerPage != null)

{

requestUrl = RequestHelper.AddQueryParameter(requestUrl, $"page-items={itemsPerPage}");

}

## Example 3

*This is an example I discovered in the existing code base.*

Technique: Decide the separator by repeated use of “**Contains**()”.

This also repeats a complex expression to obtain the target string (**this**.context.Host.RequestUrl).

public async Task SubmitRequestAsync()

{

this.context.Host.RequestUrl = $"v1/accounts/{this.AccountId}/product-instances";

if (this.PageNumber != null && this.ItemsPerPage != null)

{

this.context.Host.RequestUrl += $"?page={this.PageNumber}";

this.context.Host.RequestUrl += $"&page-items={this.ItemsPerPage}";

}

if (this.addBusinessIdToRequest)

{

string paramDelimiter = this.context.Host.RequestUrl.Contains("?") ? "&" : "?";

this.context.Host.RequestUrl += $"{paramDelimiter}businessId={this.BusinessParam}";

}

if (this.addProductLicenseIdToRequest)

{

string paramDelimiter = this.context.Host.RequestUrl.Contains("?") ? "&" : "?";

this.context.Host.RequestUrl += $"{paramDelimiter}productLicenseId={this.ProductLicenseParam}";

}

if (this.addProductTypeToRequest)

{

string paramDelimiter = this.context.Host.RequestUrl.Contains("?") ? "&" : "?";

this.context.Host.RequestUrl += $"{paramDelimiter}productType={this.ProductTypeParam}";

}

if (this.addExcludeSoftDeletedToRequest)

{

string paramDelimiter = this.context.Host.RequestUrl.Contains("?") ? "&" : "?";

this.context.Host.RequestUrl += $"{paramDelimiter}excludeSoftDeleted=false";

}

await this.context.Host.GetAsync();

}

# Proposed Syntax Example

Declarative syntax isn’t used that frequently in the code base as it stands, but to understand it, think of it as describing what the overall expression is **returning**.

This is as opposed to imperative syntax that describes operations being *performed on something*.

This is my proposed declarative style for describing a request URL, supporting **optional** parameters:

var requestUrl =

**RequestUrl**($"/v1/accounts/{accountId}/product-instances")

.**MaybeWith**("businessId", businessId)

.**MaybeWith**("productLicenseId", productLicenseId)

.**MaybeWith**("productType", productType)

.**MaybeWith**("page", page)

.**MaybeWith**("page-items", itemsPerPage)

.**When**(includeSoftDeleted).**Include**("includeSoftDeleted", "true")

.**ToString**();  
  
Interpretation:

*“The* ***requestUrl*** *is <this path> maybe with <these parameters>, and when including soft deleted items we include “includeSoftDeleted=true” too.”*

## Brief Implementation Notes

The **RequestUrl**() function would be a **static** function in a static class that takes the root URL path and returns a **builder object**. *Static classes can be subject to “****using static****” in recent C#, for a little extra syntax sugar.*

The builder object would have a collection of methods to support appending query parameters, every time, or conditionally. It would be useful to support appending Nullable types only if the nullable value is set.

The **.When(expression).Include(value)** syntax would require a *second* class, returned by **When()** which would have **Include()** as a single method. This way the Intellisense drop-lists guide the user nicely.

Generally, the methods would continue to return the builder object, until **ToString()** is called to obtain the final string.

### Enhancement for lazy evaluation

As it stands, the value-expressions are **eagerly** evaluated. Some expressions may require **lazy** evaluation, so, **.Include(f)** would be useful where f is a **Func<T>** that can be called to obtain the value to append, only when the condition is **true**.

An overload will deal with this, expressed in terms of the eager variant.

## Explanation of Proposed Functions

*These are the ones I use in the proposed codebase.*

### With()

This function provides for the case where we always desire to append a parameter.

It throws an argument exception on **null**, since this is by design a **non-optional** function. Hopefully this might trap an error in a test framework!

I propose a convenient generic overload that will perform a **ToString()** on its argument. This isn’t essential, but makes the syntax look nicer for passing e.g.: Type **int** as **i** rather than **i.ToString()**.

### MaybeWith()

This function is designed for use with **nullable value types**. It will append the parameter and value **only if** the value is **not null**. Again, it will call **ToString()** on the argument.

*Note:* It is by design that this is **not** presented as an overload of **With()**! It is my opinion that programmers use overloads too often, and this can lead to difficulty understanding precisely what will happen at the call site. In this case the word “with” implies inclusion in all cases, which is not true for an optional parameter encoded as a nullable type.

### When()

This function is the most general.

It takes a Boolean condition, and needs to be chained with a **.Include()** call, which specifies the parameter name and value to be appended.

The append takes place only if the Boolean expression is true at runtime.

The **Include()** function has a variant that accepts a lambda function, allowing the value expression to be evaluated only if the append is going to go ahead.

# Code Size Comparison – typical practice vs. proposal

The following diagram is a birds-eye view of the code to build a URL query string.

This compares current practice *vs.* proposed equivalent code.

|  |  |
| --- | --- |
| CURRENT | PROPOSED |
|  |  |

# Further Work

### Adding query parameters to a Dictionary

Similar problem. NB: **not considered in this proposal**:

Text

Description automatically generated

# References

## Portfolio Repository

The code can be found at the following location.

This is not a drop-in for the SBC framework. It could be re-worked to fit in somewhere, however.

*NB: I have omitted some homework exercises – escaping, for example. This is tangential to the point of this proposal, and needs including.*

git clone <https://github.com/jonathan-markland-at-sage/URLParameterBuilder>