Strongly Typed Constant Parameters

You will eventually run into the need for a function to accept a restricted set of value for a parameter.
One way to solve this is with an enum.

But, enums can be cast and overrode, without concern.
So, here's a more type-safe method.

This technique uses strongly-typed constants.

References

Here's some referencing articles:

Enum Alternatives in C#

Here's an article on how to store the smart enums, from the above article, using EF: Persisting the Type Safe Enum Pattern with EF 6

Ultimately, here's a nuget package that might be an end-all for us, but it hasn't been evaluated yet:

GitHub - ardalis/SmartEnum: A base class for quickly and easily creating strongly typed enum replacements in C#.

Local Implementation

Below, is how we are currently implementing strongly-typed constant parameters.

This class contains the constants and enough logic to make them work:

```
public class IFQ_MessageType
{
```

```
    public static IFQ_MessageType Chat_DeliveryAck {get;} = new IFQ_MessageType(0, "chat.deliveryack");
    public static IFQ_MessageType Chat_Invite {get;} = new IFQ_MessageType(1, "chat.invite.message");
    public static IFQ_MessageType Chat_InviteCancelled {get;} = new IFQ_MessageType(2,
"chat.invitecancelled.message");

    public string Name { get; private set; }
    public int Value { get; private set; }

    static public bool Throw_IfNotFound { get; set; } = true;

    private IFQ_MessageType(int val, string name)
    {
        Value = val;
        Name = name;
    }

    public static IEnumerable<IFQ_MessageType> List()
    {
        // Alternately, use a dictionary keyed by value...
        return new[] { Chat_DeliveryAck, Chat_Invite, Chat_InviteCancelled };
    }

    public static IFQ_MessageType FromString(string messagetypestring)
    {
        var val = List().FirstOrDefault(r => String.Equals(r.Name, messagetypestring,
StringComparison.OrdinalIgnoreCase));
        if (val == null && Throw_IfNotFound)
            throw new InvalidMessageTypeException(messagetypestring);
        return val;
    }

    public static IFQ_MessageType FromValue(int value)
    {
        var val = List().FirstOrDefault(r => r.Value == value);
        if (val == null && Throw_IfNotFound)
            throw new InvalidMessageTypeException(value.ToString());
        return val;
    }
}
```

The above class defines some constants, and their string and numeric values.

It contains FromString and FromValue methods for converting the primitive type back into the correct property.

If the FromString or FromValue cannot match its received value to a named property of the class, a null will be returned or an exception thrown (based on the state of: Throw_NotFound.

Here’s the not found exception to that can be used with the above class:

```
[Serializable]
class InvalidMessageTypeException : Exception
{
    public InvalidMessageTypeException() {  }

    public InvalidMessageTypeException(string name)
        : base(String.Format("Invalid MessageType: {0}", name))
    { }
}
```

Working Sample Usage

Here’s a working example.

The Handler class contains a method that accepts a strongly-typed parameter, and unwraps the inner value.

```
public class POCO
{
    public string Name1 { get; set; }
    public string Name2 { get; set; }
}

public class Handler
{
    public async Task<POCO> WrapPOCO(IFQ_MessageType role)
    {
        var p = new POCO();
        p.Name1 = role.Name;
        return p;
    }
```

```
}
```

And, we call the handler method, like this:

```
var hr = new Handler();
var res = await hr.WrapPOCO(IFQ_MessageType.Chat_InviteAccepted);
```

Revision #2
Created 21 April 2025 02:41:22 by glwhite
Updated 21 April 2025 02:59:14 by glwhite