Resumable Functions

A simple way to code

```
public class ClientOnboardingWorkflowPrivate : ResumableFunctionsContainer 1
{
    [ResumableFunctionEntryPoint("ClientOnboardingWorkflowPrivate.Start")] 2
    0 references
    internal async IAsyncEnumerable<Wait> StartClientOnboardingWorkflow() 3
    {
        var userId = -1;
    4   yield return
    5       Wait<RegistrationForm, RegistrationResult>(_service.ClientFillsForm, "Wait User Registration")
    6       .MatchIf((regForm, regResult) => regResult.FormId > 0)
    7       .AfterMatch((regForm, regResult) =>
            {
                FormId = regResult.FormId;
                userId = regForm.UserId;
            });

        var ownerTaskId = _service.AskOwnerToApproveClient(FormId).Id;
        var ownerDecision = false;
        yield return
            Wait<OwnerApproveClientInput, OwnerApproveClientResult>(_service.OwnerApproveClient, "Wait Owner Approve Client")
            .MatchIf((approveClientInput, _) => approveClientInput.TaskId == ownerTaskId)
            .AfterMatch((approveClientInput, _) => ownerDecision = approveClientInput.Decision);
        /*some code*/
        if (ownerDecision is false)
        {
```

Lines Description

1. A resumable function must be defined in a class that inherits from `ResumableFunctionsContainer`.

2. We add the `[ResumableFunctionEntryPoint]` attribute to the resumable function to tell the library to register or save the first wait in the database when it scans the DLL for resumable functions.

3. The resumable function must return an `IAsyncEnumerable<Wait>` and must have no input parameters.

4. Each `yield return` statement is a place where the function execution can be paused until the required wait is matched, the pause may be days or months.

```
5 Wait<RegistrationForm, RegistrationResult>(_service.ClientFillsForm, "Wait User Registration")
6 .MatchIf((regForm, regResult) => regResult.FormId > 0)
7 .AfterMatch((regForm, regResult) =>
  {
      FormId = regResult.FormId;
      userId = regForm.UserId;
  });
```

5. We tell the library that we want to wait for the method `_service.ClientFillsForm` to be executed. This method has an input of type `RegistrationForm` and an output of type `RegistrationResult`.

6. When the `ClientFillsForm` method is executed, the library will evaluate its input and output against the match expression. If the match expression is satisfied, the function execution will be resumed. Otherwise, the execution will not be resumed.

7. If we need to capture the input and output of the `ClientFillsForm` method after the match expression is satisfied, we can use the `AfterMatch` method.

The library **saves the state** of the resumable function **in** the **database**. This includes a serialized instance of the class that contains the resumable function, as well as any local variables.

The Method You Want to Wait

```
[PushCall("ClientOnboardingService.ClientFillsForm")]
6 references | 0/2 passing
public RegistrationResult ClientFillsForm(RegistrationForm registrationForm)
{
    return new RegistrationResult
    {
        FormId = Random.Shared.Next()
    };
}
```

PushCall Attribute

- The attribute [PushCall] must be added to the method you want to wait.
- The method must have one input parameter.
- This attribute will enable the method to push its input and output to the library when executed.

What Else?

- The library can wait for a method in another service which enables writing distributed services easier.
- You can wait for a group of methods.
- You can define sub resumable function and wait for it or wait for a mixed group of all types.
- You can wait for a time.
- The library has a UI to show your resumable functions and its instances.
- Resumable functions are testable this means you can write unit tests to check if it works as expected or not.