# Workflow Core

**Workflow Core** es un motor de **workflow** ligero para**.NET Standard**. Piénsese en procesos de larga duración con múltiples tareas que necesitan realizar un seguimiento del estado. Admite proveedores de persistencia y concurrencia conectables para permitir clústeres de varios nodos.

## Installing

Install the NuGet package "**WorkflowCore**"

Using **nuget**

PM> Install-Package WorkflowCore

Using .net cli

dotnet add package WorkflowCore

## Fluent API

Define workflows with the fluent API.

public class MyWorkflow : IWorkflow {

public void Build(IWorkflowBuilder<MyData> builder) {

builder

.StartWith<Task1>()

.Then<Task2>()

.Then<Task3>;

}

}

# Getting Started - Basic Concepts

## Steps

Un **workflow** consta de una serie de pasos (**steps**) conectados. Cada paso puede tener parámetros de entrada y producir salidas que se pueden devolver al **workflow** en el cual existen.

Los pasos se definen definiendo una clase que hereda de las clases abstractas **StepBody** o

**StepBodyAsync** e implementando el método **Run/RunAsync.** También se pueden crear en línea al definir el **workflow**.

### First we define some steps

public class HelloWorld : StepBody

{

public override ExecutionResult Run(IStepExecutionContext context) {

Console.WriteLine("Hello world");

return ExecutionResult.Next();

}

}

Las implementaciones de las clases **StepBody** y **StepBodyAsync** son construidas por el host del **Workflow** que primero intenta usar **IServiceProvider** a través del constructor de inyección de dependencias, si no puede construirlo con este método, buscará un constructor **sin parámetros.**

Después definimos la estructura del **workflow** mediante la composición de una cadena de **steps**. Esto se hace implementando la interfaz **IWorkflow**

public class HelloWorldWorkflow : IWorkflow {

public string Id => "HelloWorld";

public int Version => 1;

public void Build(IWorkflowBuilder<object> builder) {

builder

.StartWith<HelloWorld>;()

.Then<GoodbyeWorld>();

}

}

La interfaz ***IWorkflow*** también tiene una propiedad **readonly** **Id** y una propiedad **readonly Version.** Estos son utilizados por el **host** del **workflow** para identificar la definición de un **workflow**.

El mismo **workflow** implementado en **JSON** se vería así:

{

"Id": "HelloWorld",

"Version": 1,

"Steps": [

{

"Id": "Hello",

"StepType": "MyApp.HelloWorld, MyApp",

"NextStepId": "Bye"

},

{

"Id": "Bye",

"StepType": "MyApp.GoodbyeWorld, MyApp"

}

]

}

También puede definir sus pasos (**steps**) en línea:

public class HelloWorldWorkflow : IWorkflow

{

public string Id => "HelloWorld";

public int Version => 1;

public void Build(IWorkflowBuilder<object> builder) {

builder

.StartWith(context =>

{

Console.WriteLine(“Hello world”);

return ExecutionResult.Next();

})

.Then(context =>

{

Console.WriteLine(“Goodbye world”);

return ExecutionResult.Next();

});

}

}

Cada **workflow** en ejecución se persiste (guarda) en el proveedor de persistencia seleccionado entre cada (**Step**) ***paso***, de modo que podemos obtener su persistencia de un momento anterior en un momento dado, para continuar la ejecución. El resultado del **Step** puede indicar al **host** del **workflow** que aplace la ejecución del **workflow** hasta un momento en el tiempo futuro o en respuesta a un evento externo.

## Host

El host **Workflow** es el servicio responsable de ejecutar los **Workflow**. Para ello, sondea (**polling**) el **proveedor de persistencia** en busca de instancias de **Workflow** que estén listas para ejecutarse, las ejecuta y, a continuación, las devuelve al proveedor de persistencia para almacenadas hasta la próxima vez que se ejecuten. También es responsable de publicar **eventos** en cualquier **Workflow** de trabajo que pueda estar esperando en uno.

### Setup

Use el método de extensión ***AddWorkflow*** para ***IServiceCollection*** para configurar el host de workflow al iniciar la aplicación. De forma predeterminada, se configura con ***MemoryPersistenceProvider*** y ***SingleNodeConcurrencyProvider*** para fines de prueba. También puede configurar un proveedor de persistencia de **BD** en este punto.

services.AddWorkflow();

### Usage

Cuando se inicie la aplicación, tome el **host** del **workflow** desde el servicio de inyección de dependencias integrado ***IServiceProvider.*** Asegúrese de llamar a ***RegisterWorkflow***, para que el **host** del **workflow** conozca todos los **workflows** y, a continuación, llame a ***Start****()* para activar el grupo de subprocesos que ejecuta los **workflows**. Utilice el método ***StartWorkflow*** para iniciar una nueva instancia de un **workflow** determinado.

var host = serviceProvider.GetService<IWorkflowHost>();

host.RegisterWorkflow<HelloWorldWorkflow>();

host.Start();

host.StartWorkflow("HelloWorld", 1, null);

Console.ReadLine();

host.Stop();

## Passing data between steps

Cada paso está destinado a ser una **caja negra** (**black-box**), por lo tanto, admiten **entradas** y **salidas**. Estas **entradas** y **salidas** (parámetros de entrada y salida) se pueden asignar/definir una clase de datos que defina los datos personalizados relevantes para cada instancia de **Workflow**.

En el ejemplo siguiente veremos cómo definir **entradas** y **salidas** en un **Step**, a continuación, se muestra cómo definir un **Workflow** con una **clase tipada** para datos internos y cómo asignar las entradas y salidas a las propiedades de la clase de datos personalizada.

//Our workflow step with inputs and outputs

public class AddNumbers : StepBody {

public int Input1 { get; set; }

public int Input2 { get; set; }

public int Output { get; set; }

public override ExecutionResult Run(IStepExecutionContext context) {

Output = (Input1 + Input2);

return ExecutionResult.Next();

}

}

//Our class to define the internal data of our workflow

public class MyDataClass {

public int Value1 { get; set; }

public int Value2 { get; set; }

public int Answer { get; set; }

}

//Our workflow definition with strongly typed

// internal data and mapped inputs & outputs

public class PassingDataWorkflow : IWorkflow<MyDataClass> {

public void Build(IWorkflowBuilder<MyDataClass> builder) {

builder

.StartWith<AddNumbers>()

.Input(step => step.Input1, data => data.Value1)

.Input(step => step.Input2, data => data.Value2)

.Output(data => data.Answer, step => step.Output)

.Then<CustomMessage>()

.Input(step => step.Message,

data => "The answer is " + data.Answer.ToString());

}

...

}

o en formato **JSON**

{

"Id": "AddWorkflow",

"Version": 1,

"DataType": "MyApp.MyDataClass, MyApp",

"Steps": [

{

"Id": "Add",

"StepType": "MyApp.AddNumbers, MyApp",

"NextStepId": "ShowResult",

"Inputs": {

"Input1": "data.Value1",

"Input2": "data.Value2"

},

"Outputs": {

"Answer": "step.Output"

}

},

{

"Id": "ShowResult",

"StepType": "MyApp.CustomMessage, MyApp",

"Inputs": {

"Message": "\"The answer is \" + data.Answer"

}

}

]

}

o en formato **YAML**

Id: AddWorkflow

Version: 1

DataType: MyApp.MyDataClass, MyApp

Steps:

- Id: Add

StepType: MyApp.AddNumbers, MyApp

NextStepId: ShowResult

Inputs:

Input1: data.Value1

Input2: data.Value2

Outputs:

Answer: step.Output

- Id: ShowResult

StepType: MyApp.CustomMessage, MyApp

Inputs:

Message: '"The answer is " + data.Answer'

## Injecting dependencies into steps

Si registra las clases de **Step** con el contenedor de **IoC**, el **host** de **workflow** usará el contenedor de **IoC** para construirlas y, por lo tanto, inyectará las dependencias necesarias.

En este ejemplo se ilustra el uso de la inyección de dependencias para los **Steps** del **workflow**.

#### Considere el siguiente servicio

public interface IMyService

{

void DoTheThings();

}

...

public class MyService : IMyService

{

public void DoTheThings()

{

Console.WriteLine("Doing stuff...");

}

}

El cual es consumido por un paso (**step**) del **workflow** de la siguiente manera:

public class DoSomething : StepBody

{

private IMyService \_myService;

public DoSomething(IMyService myService) {

\_myService = myService;

}

public override ExecutionResult Run(IStepExecutionContext context) {

\_myService.DoTheThings();

return ExecutionResult.Next();

}

}

Simplemente agregaremos ambos al servicio, el **Step** y el **Workflow** como **transitorios** a la colección de servicios al configurar su contenedor de **IoC**.

(Evite registrar **Steps** como **singletons**, ya que es posible que varios flujos de trabajo simultáneos deban usarlos a la vez).

IServiceCollection services = new ServiceCollection();

services.AddLogging();

services.AddWorkflow();

services.AddTransient<DoSomething>();

services.AddTransient<IMyService, MyService>();

# External events -Events

Un **workflow** también puede esperar un **evento**/**notificación** externa antes de continuar. En el ejemplo siguiente, el **workflow** esperará un evento denominado *"****MyEvent****"* con una clave de ***0***. Una vez que una fuente externa ha activado este evento, el flujo de trabajo se activará y continuará procesando, pasando los datos generados por el evento al siguiente **Step**.

public class EventSampleWorkflow : IWorkflow<MyDataClass> {

public void Build(IWorkflowBuilder<MyDataClass> builder) {

builder

.StartWith(context => ExecutionResult.Next())

.WaitFor("MyEvent", data => "0")

.Output(data => data.Value, step => step.EventData)

.Then<CustomMessage>()

.Input(step => step.Message,

data => "The data from the event is " + data.Value);

}

}

...

//External events are published via the host

//All workflows that have subscribed to MyEvent 0, will be passed "hello"

host.PublishEvent("MyEvent", "0", "hello");

## Effective Date

También puede especificar una fecha efectiva o de entrada en vigor, cuando esperamos eventos, lo que nos permite responder a eventos que pueden haber ocurrido en el pasado o solo a los que ocurren después de la fecha efectiva o de entrada en vigor.

## JSON / YAML API

El **.WaitFor** puede ser implementado utilizando los **inputs** como sigue:

| **Field** | **Descripción** |
| --- | --- |
| **CancelCondition** | **Optional expression to specify a cancel condition** |
| **Inputs.EventName** | **Expression to specify the event name** |
| **Inputs.EventKey** | **Expression to specify the event key** |
| **Inputs.EffectiveDate** | **Optional expression to specify the effective date** |

{

"Id": "MyWaitStep",

"StepType": "WorkflowCore.Primitives.WaitFor, WorkflowCore",

"NextStepId": "...",

"CancelCondition": "...",

"Inputs": {

"EventName": "\"Event1\"",

"EventKey": "\"Key1\"",

"EffectiveDate": "DateTime.Now"

}

}

Id: MyWaitStep

StepType: WorkflowCore.Primitives.WaitFor, WorkflowCore

NextStepId: "..."

CancelCondition: "..."

Inputs:

EventName: '"Event1"'

EventKey: '"Key1"'

EffectiveDate: DateTime.Now

# Activity workers - Activities

Una actividad se define como un **item** de una cola de trabajo externa, al que un **workflow** puede esperar.

En este ejemplo, el **workflow** esperará a **la actividad-1**, antes de continuar. También pasa el valor de los **data.Value1** a la actividad, luego asigna el resultado de la actividad a los **data.Value2**.

Luego creamos un **worker** para procesar la cola de elementos de actividad. Utiliza el método **GetPendingActivity** para obtener la actividad pendiente y los datos que un **workflow** está esperando.

public class ActivityWorkflow : IWorkflow<MyData> {

public void Build(IWorkflowBuilder<MyData> builder)

{

builder

.StartWith<HelloWorld>()

.Activity("activity-1", (data) => data.Value1)

.Output(data => data.Value2, step => step.Result)

.Then<PrintMessage>()

.Input(step => step.Message, data => data.Value2);

}

}

...

var activity = host.GetPendingActivity("activity-1", "worker1", TimeSpan.FromMinutes(1)).Result;

if (activity != null) {

Console.WriteLine(activity.Parameters);

host.SubmitActivitySuccess(activity.Token, "Some response data");

}

The **JSON** representation of this step would look like this

{

"Id": "activity-step",

"StepType": "WorkflowCore.Primitives.Activity, WorkflowCore",

"Inputs":

{

"ActivityName": "\"activity-1\"",

"Parameters": "data.Value1"

},

"Outputs": { "Value2": "step.Result" }

}

## JSON / YAML API

The Activity step can be configured using inputs as follows

El **Step** **Activity** puede ser configurado utilizando entradas como las siguientes

| **Field** | **Description** |  |
| --- | --- | --- |
| CancelCondition | Optional expression to specify a cancel condition |  |
| Inputs.ActivityName | Expression to specify the activity name |  |
| Inputs.Parameters | Expression to specify the parameters to pass the activity worker |  |
| Inputs.EffectiveDate | Optional expression to specify the effective date |  |

{

"Id": "MyActivityStep",

"StepType": "WorkflowCore.Primitives.Activity, WorkflowCore",

"NextStepId": "...",

"CancelCondition": "...",

"Inputs": {

"ActivityName": "\"my-activity\"",

"Parameters": "data.SomeValue"

}

}

Id: MyActivityStep

StepType: WorkflowCore.Primitives.Activity, WorkflowCore

NextStepId: "..."

CancelCondition: "..."

Inputs:

ActivityName: '"my-activity"'

EventKey: '"Key1"'

Parameters: data.SomeValue

# Error handling

Cada **step** se puede configurar con su propio comportamiento de manejo de errores, se puede volver a intentar en un momento posterior, suspender el flujo de trabajo o finalizar el flujo de trabajo.

## Fluent API

public void Build(IWorkflowBuilder<object> builder)

{

builder

.StartWith<HelloWorld>()

.OnError(WorkflowErrorHandling.Retry, TimeSpan.FromMinutes(10))

.Then<GoodbyeWorld>();

}

### JSON / YAML API

**ErrorBehavior**

{

"Id": "...",

"StepType": "...",

"ErrorBehavior": "Retry / Suspend / Terminate / Compensate",

"RetryInterval": "00:10:00"

}

Id: "..."

StepType: "..."

ErrorBehavior: Retry / Suspend / Terminate / Compensate

RetryInterval: '00:10:00'

## Global Error handling

The **WorkflowHost** service also has a .**OnStepError** event which can be used to intercept exceptions from **workflow** steps on a more global level.

# Control Structures

## Decision Branches

You can define multiple independent branches within your workflow and select one based on an expression value.

#### Fluent API

Para la **API** fluida, definimos nuestras ramas con el método **CreateBranch()** en el generador de flujo de trabajo. A continuación, podemos seleccionar una rama utilizando el método **Branch**.

The select expressions will be matched to the branch listed via the Branch method, and the matching next step(s) will be scheduled to execute next. **Matching** multiple next steps will result in parallel branches running.

Las expresiones seleccionadas se emparejarán con la **branch** enumerada a través del método **Branch**, y los siguientes pasos coincidentes se programarán para ejecutarse a continuación. Hacer coincidir varios pasos siguientes dará como resultado la ejecución de ramas paralelas.

This workflow will select branch1 if the value of **data.Value1** is one, and branch2 if it is two.

Este flujo de trabajo seleccionará la rama1 si el valor de los **data.Value1** es uno, y **branch2** si es dos.

var branch1 = builder.CreateBranch()

.StartWith<PrintMessage>()

.Input(step => step.Message, data => "hi from 1")

.Then<PrintMessage>()

.Input(step => step.Message, data => "bye from 1");

var branch2 = builder.CreateBranch()

.StartWith<PrintMessage>()

.Input(step => step.Message, data => "hi from 2")

.Then<PrintMessage>()

.Input(step => step.Message, data => "bye from 2");

Builder

.StartWith<HelloWorld>()

.Decide(data => data.Value1)

.Branch((data, outcome) => data.Value1 == "one", branch1)

.Branch((data, outcome) => data.Value1 == "two", branch2);

#### JSON / YAML API

Hook up your branches via the **SelectNextStep** property, instead of a **NextStepId**. The expressions will be matched to the step Ids listed in **SelectNextStep**, and the matching next step(s) will be scheduled to execute next.

Conecte sus ramas a través de la propiedad **SelectNextStep,** en lugar de un **NextStepId**. Las expresiones coincidirán con los identificadores de paso enumerados en **SelectNextStep** y los siguientes pasos coincidentes se programarán para ejecutarse a continuación.

{

"Id": "DecisionWorkflow",

"Version": 1,

"DataType": "MyApp.MyData, MyApp",

"Steps": [

{

"Id": "decide",

"StepType": "...",

"SelectNextStep":

{

"Branch1": "<<result expression to match for branch 1>>",

"Branch2": "<<result expression to match for branch 2>>"

}

},

{

"Id": "Branch1",

"StepType": "MyApp.PrintMessage, MyApp",

"Inputs":

{

"Message": "\"Hello from 1\""

}

},

{

"Id": "Branch2",

"StepType": "MyApp.PrintMessage, MyApp",

"Inputs":

{

"Message": "\"Hello from 2\""

}

}

]

}

Id: DecisionWorkflow

Version: 1

DataType: MyApp.MyData, MyApp

Steps:

- Id: decide

StepType: WorkflowCore.Primitives.Decide, WorkflowCore

Inputs:

Expression: <<input expression to evaluate>>

OutcomeSteps:

Branch1: '<<result expression to match for branch 1>>'

Branch2: '<<result expression to match for branch 2>>'

- Id: Branch1

StepType: MyApp.PrintMessage, MyApp

Inputs:

Message: '"Hello from 1"'

- Id: Branch2

StepType: MyApp.PrintMessage, MyApp

Inputs:

Message: '"Hello from 2"'

## Parallel ForEach

Use el método **.ForEach** para iniciar un bucle paralelo **For**

#### Fluent API

public class ForEachWorkflow : IWorkflow {

public string Id => "Foreach";

public int Version => 1;

public void Build(IWorkflowBuilder<object> builder) {

builder

.StartWith<SayHello>()

.ForEach(data => new List<int>() { 1, 2, 3, 4 })

.Do(x => x

.StartWith<DisplayContext>()

.Input(step => step.Message, (data, context) => context.Item)

.Then<DoSomething>())

.Then<SayGoodbye>();

}

}

#### JSON / YAML API

{

"Id": "MyForEachStep",

"StepType": "WorkflowCore.Primitives.ForEach, WorkflowCore",

"NextStepId": "...",

"Inputs": { "Collection": "<<expression to evaluate>>" },

"Do": [[

{

"Id": "do1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "do2"

},

{

"Id": "do2",

"StepType": "MyApp.DoSomething2, MyApp"

}

]]

}

Id: MyForEachStep

StepType: WorkflowCore.Primitives.ForEach, WorkflowCore

NextStepId: "..."

Inputs:

Collection: "<<expression to evaluate>>"

Do:

- - Id: do1

StepType: MyApp.DoSomething1, MyApp

NextStepId: do2

- Id: do2

StepType: MyApp.DoSomething2, MyApp

## While Loops

Use the **.While** method to start a while construct

#### Fluent API

public class WhileWorkflow : IWorkflow<MyData>

{

public string Id => "While";

public int Version => 1;

public void Build(IWorkflowBuilder<MyData> builder){

builder

.StartWith<SayHello>()

.While(data => data.Counter < 3)

.Do(x => x

.StartWith<DoSomething>()

.Then<IncrementStep>()

.Input(step => step.Value1, data => data.Counter)

.Output(data => data.Counter, step => step.Value2))

.Then<SayGoodbye>();

}

}

#### JSON / YAML API

{

"Id": "MyWhileStep",

"StepType": "WorkflowCore.Primitives.While, WorkflowCore",

"NextStepId": "...",

"Inputs": { "Condition": "<<expression to evaluate>>" },

"Do": [[

{

"Id": "do1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "do2"

},

{

"Id": "do2",

"StepType": "MyApp.DoSomething2, MyApp"

}

]]

}

Id: MyWhileStep

StepType: WorkflowCore.Primitives.While, WorkflowCore

NextStepId: "..."

Inputs:

Condition: "<<expression to evaluate>>"

Do:

- - Id: do1

StepType: MyApp.DoSomething1, MyApp

NextStepId: do2

- Id: do2

StepType: MyApp.DoSomething2, MyApp

## If Conditions

Use the .If method to start an if condition

#### Fluent API

public class IfWorkflow : IWorkflow<MyData>

{

public void Build(IWorkflowBuilder<MyData> builder) {

builder

.StartWith<SayHello>()

.If(data => data.Counter < 3).Do(then => then

.StartWith<PrintMessage>()

.Input(step => step.Message, data => "Value is less than 3")

)

.If(data => data.Counter < 5).Do(then => then

.StartWith<PrintMessage>()

.Input(step => step.Message, data => "Value is less than 5")

)

.Then<SayGoodbye>();

}

}

#### JSON / YAML API

{

"Id": "MyIfStep",

"StepType": "WorkflowCore.Primitives.If, WorkflowCore",

"NextStepId": "...",

"Inputs": { "Condition": "<<expression to evaluate>>" },

"Do": [[

{

"Id": "do1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "do2"

},

{

"Id": "do2",

"StepType": "MyApp.DoSomething2, MyApp"

}

]]

}

Id: MyIfStep

StepType: WorkflowCore.Primitives.If, WorkflowCore

NextStepId: "..."

Inputs:

Condition: "<<expression to evaluate>>"

Do:

- - Id: do1

StepType: MyApp.DoSomething1, MyApp

NextStepId: do2

- Id: do2

StepType: MyApp.DoSomething2, MyApp

#### Parallel Paths

Utilice el método **.Parallel()**  para ramificar tareas paralelas

#### Fluent API

public class ParallelWorkflow : IWorkflow<MyData> {

public string Id => "parallel-sample";

public int Version => 1;

public void Build(IWorkflowBuilder<MyData> builder) {

builder

.StartWith<SayHello>()

.Parallel()

.Do(then => then.StartWith<Task1dot1>().Then<Task1dot2>()

.Do(then => then.StartWith<Task2dot1>().Then<Task2dot2>()

.Do(then => then.StartWith<Task3dot1>().Then<Task3dot2>()

.Join()

.Then<SayGoodbye>();

}

}

#### JSON / YAML API

{

"Id": "MyParallelStep",

"StepType": "WorkflowCore.Primitives.Sequence, WorkflowCore",

"NextStepId": "...",

"Do": [

[

{

"Id": "Branch1.Step1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "Branch1.Step2"

},

{

"Id": "Branch1.Step2",

"StepType": "MyApp.DoSomething2, MyApp"

}

],

[

{

"Id": "Branch2.Step1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "Branch2.Step2"

},

{

"Id": "Branch2.Step2",

"StepType": "MyApp.DoSomething2, MyApp"

}

]

]

}

Id: MyParallelStep

StepType: WorkflowCore.Primitives.Sequence, WorkflowCore

NextStepId: "..."

Do:

- - Id: Branch1.Step1

StepType: MyApp.DoSomething1, MyApp

NextStepId: Branch1.Step2

- Id: Branch1.Step2

StepType: MyApp.DoSomething2, MyApp

- - Id: Branch2.Step1

StepType: MyApp.DoSomething1, MyApp

NextStepId: Branch2.Step2

- Id: Branch2.Step2

StepType: MyApp.DoSomething2, MyApp

## Schedule

Usamos **.Schedule** para registrar un conjunto futuro de pasos para que se ejecuten de forma asincrónica en segundo plano dentro del flujo de trabajo.

#### Fluent API

builder

.StartWith(

context => Console.WriteLine("Hello"))

.Schedule(data => TimeSpan.FromSeconds(5))

.Do(schedule =>

schedule.StartWith(context => Console.WriteLine("Doing scheduled tasks"))

)

.Then(context => Console.WriteLine("Doing normal tasks"));

#### JSON / YAML API

{

"Id": "MyScheduleStep",

"StepType": "WorkflowCore.Primitives.Schedule, WorkflowCore",

"Inputs": { "Interval": "<<expression to evaluate>>" },

"Do": [[

{

"Id": "do1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "do2"

},

{

"Id": "do2",

"StepType": "MyApp.DoSomething2, MyApp"

}

]]

}

Id: MyScheduleStep

StepType: WorkflowCore.Primitives.Schedule, WorkflowCore

Inputs:

Interval: "<<expression to evaluate>>"

Do:

- - Id: do1

StepType: MyApp.DoSomething1, MyApp

NextStepId: do2

- Id: do2

StepType: MyApp.DoSomething2, MyApp

### Delay

The **Delay** step will pause the current branch of your workflow for a specified period.

El Step **Delay** pausará la rama actual del flujo de trabajo durante un período especificado.

#### JSON / YAML API

{

"Id": "MyDelayStep",

"StepType": "WorkflowCore.Primitives.Delay, WorkflowCore",

"NextStepId": "...",

"Inputs": { "Period": "<<expression to evaluate>>" }

}

Id: MyDelayStep

StepType: WorkflowCore.Primitives.Delay, WorkflowCore

NextStepId: "..."

Inputs:

Period: "<<expression to evaluate>>"

### Recur

Utilice **.Recur** para configurar un conjunto de pasos en segundo plano recurrentes dentro de su workflow, hasta que se cumpla una determinada condición

#### Fluent API

builder

.StartWith(context => Console.WriteLine("Hello"))

.Recur(data => TimeSpan.FromSeconds(5), data => data.Counter > 5).Do(recur => recur

.StartWith(context => Console.WriteLine("Doing recurring task"))

)

.Then(context => Console.WriteLine("Carry on"));

#### JSON / YAML API

{

"Id": "MyScheduleStep",

"StepType": "WorkflowCore.Primitives.Recur, WorkflowCore",

"Inputs": {

"Interval": "<<expression to evaluate>>",

"StopCondition": "<<expression to evaluate>>"

},

"Do": [[

{

"Id": "do1",

"StepType": "MyApp.DoSomething1, MyApp",

"NextStepId": "do2"

},

{

"Id": "do2",

"StepType": "MyApp.DoSomething2, MyApp"

}

]]

}

Id: MyScheduleStep

StepType: WorkflowCore.Primitives.Recur, WorkflowCore

Inputs:

Interval: "<<expression to evaluate>>"

StopCondition: "<<expression to evaluate>>"

Do:

- - Id: do1

StepType: MyApp.DoSomething1, MyApp

NextStepId: do2

- Id: do2

StepType: MyApp.DoSomething2, MyApp

# Saga transaction with compensation

Una **Saga** nos permite encapsular una secuencia de pasos dentro de una **transacción saga** y definir los **Steps** (pasos) de compensación para cada uno de ellos.

En el ejemplo, **Task2** generará una excepción, por lo que se lanzarán **UndoTask2** y **UndoTask1**.

builder

.StartWith(context => Console.WriteLine("Begin"))

.Saga(saga => saga

.StartWith<Task1>().CompensateWith<UndoTask1>()

.Then<Task2>().CompensateWith<UndoTask2>()

.Then<Task3>().CompensateWith<UndoTask3>()

)

.CompensateWith<CleanUp>()

.Then(context => Console.WriteLine("End"));

## Retry policy for failed saga transaction

Este ejemplo en particular volverá a intentar la saga cada 5 segundos, pero también podría simplemente fallar por completo y procesar una tarea de compensación maestra para toda la saga.

builder

.StartWith(context => Console.WriteLine("Begin"))

.Saga(saga => saga

.StartWith<Task1>()

.CompensateWith<UndoTask1>()

.Then<Task2>()

.CompensateWith<UndoTask2>()

.Then<Task3>()

.CompensateWith<UndoTask3>()

)

.OnError(Models.WorkflowErrorHandling.Retry, TimeSpan.FromSeconds(5))

.Then(context => Console.WriteLine("End"));

## Compensate entire saga transaction

También puede especificar solo un paso de compensación maestro, de la siguiente manera:

builder

.StartWith(context => Console.WriteLine("Begin"))

.Saga(saga => saga

.StartWith<Task1>()

.Then<Task2>()

.Then<Task3>()

)

.CompensateWith<UndoEverything>()

.Then(context => Console.WriteLine("End"));

## Passing parameters to compensation steps

Los parámetros se pueden pasar a un **Step** (paso) de compensación de la siguiente manera:

builder

.StartWith<SayHello>()

.CompensateWith<PrintMessage>(compensate =>

{

compensate.Input(step => step.Message, data => "undoing...");

})

## Expressing a saga in JSON or YAML

A saga transaction can be expressed in JSON or YAML, by using the **WorkflowCore.Primitives.Sequence** **step** and setting the **Saga** parameter to **true**.

The compensation steps can be defined by specifying the **CompensateWith** parameter.

Una **transacción** **saga** se puede expresar en **JSON** o **YAML**, utilizando el **Step** **WorkflowCore.Primitives.Sequence** y estableciendo el parámetro **Saga** en **true**.

Los pasos de compensación se pueden definir especificando el parámetro **CompensateWith.**

{

"Id": "Saga-Sample",

"Version": 1,

"DataType": "MyApp.MyDataClass, MyApp",

"Steps": [

{

"Id": "Hello",

"StepType": "MyApp.HelloWorld, MyApp",

"NextStepId": "MySaga"

},

{

"Id": "MySaga",

"StepType": "WorkflowCore.Primitives.Sequence, WorkflowCore",

"NextStepId": "Bye",

"Saga": true,

"Do": [

[

{

"Id": "do1",

"StepType": "MyApp.Task1, MyApp",

"NextStepId": "do2",

"CompensateWith": [

{

"Id": "undo1",

"StepType": "MyApp.UndoTask1, MyApp"

}

]

},

{

"Id": "do2",

"StepType": "MyApp.Task2, MyApp",

"CompensateWith": [

{

"Id": "undo2-1",

"NextStepId": "undo2-2",

"StepType": "MyApp.UndoTask2, MyApp"

},

{

"Id": "undo2-2",

"StepType": "MyApp.DoSomethingElse, MyApp"

}

]

}

]

]

},

{

"Id": "Bye",

"StepType": "MyApp.GoodbyeWorld, MyApp"

}

]

}

Id: Saga-Sample

Version: 1

DataType: MyApp.MyDataClass, MyApp

Steps:

- Id: Hello

StepType: MyApp.HelloWorld, MyApp

NextStepId: MySaga

- Id: MySaga

StepType: WorkflowCore.Primitives.Sequence, WorkflowCore

NextStepId: Bye

Saga: true

Do:

- - Id: do1

StepType: MyApp.Task1, MyApp

NextStepId: do2

CompensateWith:

- Id: undo1

StepType: MyApp.UndoTask1, MyApp

- Id: do2

StepType: MyApp.Task2, MyApp

CompensateWith:

- Id: undo2-1

NextStepId: undo2-2

StepType: MyApp.UndoTask2, MyApp

- Id: undo2-2

StepType: MyApp.DoSomethingElse, MyApp

- Id: Bye

StepType: MyApp.GoodbyeWorld, MyApp

# JSON / YAML Definitions - Loading workflow definitions from JSON or YAML

Instale **WorkflowCore**.**DSL** desde **nuget** y llame a **AddWorkflowDSL** en su colección de servicios. A continuación, tome el **DefinitionLoader** del contenedor de **IoC** y llame al método. **LoadDefinition**

using WorkflowCore.Interface;

...

var loader = serviceProvider.GetService<IDefinitionLoader>();

loader.LoadDefinition("<<json or yaml string here>>", Deserializers.Json);

## Common DSL

Both the **JSON and YAML** formats follow a common **DSL**, where **step** types within the **workflow** are referenced by the fully qualified class names. Built-in step types typically live in the **WorklfowCore.Primitives** namespace.

Tanto los formatos **JSON** como **YAML** siguen un **DSL** común, donde los tipos de **Steps** dentro del **workflow** son referenciados por los nombres de clase completos. Los tipos de **Steps** integrados suelen vivir en el espacio **de nombres WorklfowCore.Primitives.**

| **Field** | **Description** |
| --- | --- |
| **Id** | **Workflow** Definition **ID** |
| **Version** | **Workflow** Definition **Version** |
| **DataType** | **Fully qualified** assembly **class** **name** of the custom data object |
| **Steps[].Id** | **Step ID (required unique key for each step)** |
| **Steps[].StepType** | **Fully qualified** assembly **class name** of the **step** |
| **Steps[].NextStepId** | **Step ID** of the next step after this one completes |
| **Steps[].Inputs** | Optional Key/value pair of step inputs |
| **Steps[].Outputs** | Optional Key/value pair of step outputs |
| **Steps[].CancelCondition** | Optional cancel condition |

{

"Id": "HelloWorld",

"Version": 1,

"Steps": [

{

"Id": "Hello",

"StepType": "MyApp.HelloWorld, MyApp",

"NextStepId": "Bye"

},

{

"Id": "Bye",

"StepType": "MyApp.GoodbyeWorld, MyApp"

}

]

}

Id: HelloWorld

Version: 1

Steps:

- Id: Hello

StepType: MyApp.HelloWorld, MyApp

NextStepId: Bye

- Id: Bye

StepType: MyApp.GoodbyeWorld, MyApp

### Inputs and Outputs

Los **inputs** y **outputs** se pueden enlazar a un Step como un objeto de par **nombre**/**valor** (clave/valor),

The **Inputs** collection, the key would match a property on the **Step** class and the value would be an expression with both the data and context parameters at your disposal.

La colección **Inputs,** la clave coincidiría con una propiedad de la clase **Step** y el valor sería una expresión con los parámetros de datos y contexto a su disposición.

The **Outputs** collection, the key would match a property on the **Data** class and the value would be an expression with both the step as a parameter at your disposal.

La colección de **Outputs,** la clave coincidiría con una propiedad de la clase **Data** y el valor sería una expresión con el paso como parámetro a su disposición.

Los detalles completos de las capacidades de expresión del lenguaje se pueden encontrar [aquí](https://github.com/StefH/System.Linq.Dynamic.Core/wiki/Dynamic-Expressions#expression-language)

<https://github.com/StefH/System.Linq.Dynamic.Core/wiki/Dynamic-Expressions#expression-language>

{

"Id": "AddWorkflow",

"Version": 1,

"DataType": "MyApp.MyDataClass, MyApp",

"Steps": [

{

"Id": "Hello",

"StepType": "MyApp.HelloWorld, MyApp",

"NextStepId": "Add"

},

{

"Id": "Add",

"StepType": "MyApp.AddNumbers, MyApp",

"NextStepId": "Bye",

"Inputs": {

"Value1": "data.Value1",

"Value2": "data.Value2"

},

"Outputs": {

"Answer": "step.Result"

}

},

{

"Id": "Bye",

"StepType": "MyApp.GoodbyeWorld, MyApp"

}

]

}

Id: AddWorkflow

Version: 1

DataType: MyApp.MyDataClass, MyApp

Steps:

- Id: Hello

StepType: MyApp.HelloWorld, MyApp

NextStepId: Add

- Id: Add

StepType: MyApp.AddNumbers, MyApp

NextStepId: Bye

Inputs:

Value1: data.Value1

Value2: data.Value2

Outputs:

Answer: step.Result

- Id: Bye

StepType: MyApp.GoodbyeWorld, MyApp

{

"Id": "AddWorkflow",

"Version": 1,

"DataType": "MyApp.MyDataClass, MyApp",

"Steps": [

{

"Id": "Hello",

"StepType": "MyApp.HelloWorld, MyApp",

"NextStepId": "Print"

},

{

"Id": "Print",

"StepType": "MyApp.PrintMessage, MyApp",

"Inputs": { "Message": "\"Hi there!\"" }

}

]

}

Id: AddWorkflow

Version: 1

DataType: MyApp.MyDataClass, MyApp

Steps:

- Id: Hello

StepType: MyApp.HelloWorld, MyApp

NextStepId: Print

- Id: Print

StepType: MyApp.PrintMessage, MyApp

Inputs:

Message: '"Hi there!"'

You can also pass object **graphs** to step inputs as opposed to just scalar values

También puede pasar grafos de objetos como **imputs** del **Step** en lugar de solo valores escalares

"inputs":

{

"Body": {

"Value1": 1,

"Value2": 2

},

"Headers": {

"Content-Type": "application/json"

}

},

Si desea evaluar una expresión para una propiedad determinada de nuestro objeto, simplemente anteponga y **@** delante del nombre de la variable y establezca una expresión en la cadena.

"inputs":

{

"Body": {

"@Value1": "data.MyValue \* 2",

"Value2": 5

},

"Headers": {

"Content-Type": "application/json"

}

},

#### Enums

Si su paso tiene una propiedad **enum,** puede pasar la representación de cadena del valor de **enum** y se convertirá automáticamente.

#### Environment variables available in input expressions

Puede acceder a las variables de entorno desde las expresiones de entrada.

#### Uso:

environment["VARIABLE\_NAME"]

# Persistence

Dado que **workflows** suelen ser procesos de larga duración, deberán persistirse (su estado) en entre pasos. Hay varios proveedores de persistencia disponibles como paquetes **Nuget** .

* **MemoryPersistenceProvider** (Default provider, for demo and testing purposes)
* [MongoDB](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Persistence.MongoDB)
* [SQL Server](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Persistence.SqlServer)
* [PostgreSQL](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Persistence.PostgreSQL)
* [Sqlite](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Persistence.Sqlite)
* [Amazon DynamoDB](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.AWS)
* [Cosmos DB](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.Azure)
* [Redis](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.Redis)

# Workflow Middleware

Los **Workflows** se pueden ampliar con **Middleware** que se ejecuta antes/después de que los **Workflows** comiencen/completen, así como en torno a los pasos del **Workflows** para proporcionar flexibilidad en la implementación de **preocupaciones transversales (cross-cutting concerns)**, como los [log correlation](https://www.frakkingsweet.com/net-core-log-correlation-easy-access-to-headers/), [retries](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly) y otros casos de uso.

Esto se hace implementando y registrando **IWorkflowMiddleware** para **Workflows** o **IWorkflowStepMiddleware** para **Steps**.

## Step Middleware

El **Step** **middleware** le permite ejecutar código adicional en torno a la ejecución de un paso determinado y alterar su comportamiento. La implementación de un **middleware** debería resultarnos familiar para cualquier persona que este familiarizada con [la canalización de middleware de ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1) o [el middleware](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-requests?view=aspnetcore-3.1#outgoing-request-middleware) [DelegatingHandler](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-requests?view=aspnetcore-3.1#outgoing-request-middleware) de [HttpClient.](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-requests?view=aspnetcore-3.1#outgoing-request-middleware)

### Usage

Primero, cree su propia clase de **middleware** que implemente **IWorkflowStepMiddleware.** Este es un ejemplo de un **middleware** que agrega el **workflow** **ID** y and **step** **ID** al contexto de log de correlación de registros de cada Step del workflow de la aplicación.

**Importante:** Debe asegurarse de llamar a **next()** como parte de su **middleware.** Si no lo hace, su paso nunca se ejecutará.

public class LogCorrelationStepMiddleware : IWorkflowStepMiddleware

{

private readonly ILogger<LogCorrelationStepMiddleware> \_log;

public LogCorrelationStepMiddleware(

ILogger<LogCorrelationStepMiddleware> log)

{

\_log = log;

}

public async Task<ExecutionResult> HandleAsync(

IStepExecutionContext context,

IStepBody body,

WorkflowStepDelegate next)

{

var workflowId = context.Workflow.Id;

var stepId = context.Step.Id;

// Uses log scope to add a few attributes to the scope

using (\_log.BeginScope(“{@WorkflowId}”, workflowId))

using (\_log.BeginScope(“{@StepId}”, stepId))

{

// Calling next ensures step gets executed

return await next();

}

}

}

Este es otro ejemplo de un **middleware** que utiliza la biblioteca de resiliencia [Polly](https://github.com/App-vNext/Polly) para implementar reintentos en los **Steps** del **workflow** basados en una directiva de reintento personalizada.

public class PollyRetryStepMiddleware : IWorkflowStepMiddleware

{

private const string StepContextKey = "WorkflowStepContext";

private const int MaxRetries = 3;

private readonly ILogger<PollyRetryStepMiddleware> \_log;

public PollyRetryMiddleware(ILogger<PollyRetryStepMiddleware> log)

{

\_log = log;

}

// Consult Polly's docs for more information on how to build

// retry policies:

// https://github.com/App-vNext/Polly

public IAsyncPolicy<ExecutionResult> GetRetryPolicy() =>

Policy<ExecutionResult>

.Handle<TimeoutException>()

.RetryAsync(

MaxRetries,

(result, retryCount, context) =>

UpdateRetryCount(

result.Exception,

retryCount,

context[StepContextKey] as IStepExecutionContext)

);

public async Task<ExecutionResult> HandleAsync(

IStepExecutionContext context,

IStepBody body,

WorkflowStepDelegate next

)

{

return await GetRetryPolicy().ExecuteAsync(

ctx => next(),

// The step execution context gets passed down so that

// the step is accessible within the retry policy

new Dictionary<string, object>

{

{ StepContextKey, context }

});

}

private Task UpdateRetryCount(

Exception exception,

int retryCount,

IStepExecutionContext stepContext)

{

var stepInstance = stepContext.ExecutionPointer;

stepInstance.RetryCount = retryCount;

return Task.CompletedTask;

}

}

## Pre/Post Workflow Middleware

**El Workflow** **middleware** se ejecuta antes de que se inicie un **workflow** o después de que se complete un **workflow** y se puede usar para conectarse al ciclo de vida del flujo de trabajo o alterar el flujo de trabajo en sí antes de que se inicie.

### Pre Workflow Middleware

Este **middleware** se ejecuta antes de que se inicie el workflow y potencialmente puede alterar las propiedades de **WorkflowInstance.**

En el ejemplo siguiente se muestra cómo establecer la propiedad **Description** en **WorkflowInstance** mediante un **middleware** que interpreta los datos del **workflow** proporcionado. Esto es útil en los casos en que desea que la descripción del flujo de trabajo se derive de los datos pasados al flujo de trabajo.

Tenga en cuenta que sea ha utilizado **WorkflowMiddlewarePhase.PreWorkflow** para especificar que se ejecute antes de que se inicie el workflow.

**Important**: You should call next as part of the workflow middleware to ensure that the next workflow in the chain runs.

**Importante:** Se debe llamar a **next()** como parte del **middleware** del **workflow** para asegurarse de que se ejecuta el siguiente **workflow** que se encuentra en la cadena de **workflows**.

// AddDescriptionWorkflowMiddleware.cs

public class AddDescriptionWorkflowMiddleware : IWorkflowMiddleware

{

public WorkflowMiddlewarePhase Phase =>

WorkflowMiddlewarePhase.PreWorkflow;

public Task HandleAsync(WorkflowInstance workflow, WorkflowDelegate next){

if (workflow.Data is IDescriptiveWorkflowParams descriptiveParams) {

workflow.Description = descriptiveParams.Description;

}

return next();

}

}

// IDescriptiveWorkflowParams.cs

public interface IDescriptiveWorkflowParams{

string Description { get; }

}

// MyWorkflowParams.cs

public MyWorkflowParams : IDescriptiveWorkflowParams {

public string Description => $"Run task ''";

public string TaskName { get; set; }

}

### Exception Handling in Pre Workflow Middleware

La gestión de *excepciones* en **el Pre-Workflow middleware**, trata de manera diferente al **Post-Workflow** **middleware**.

Dado que el **middleware** se ejecuta antes de que se inicie el **workflow,** cualquier excepción lanzada dentro de un **Pre-workflow** **middleware** se acumulará (**bubble up**) hasta el método **StartWorkflow** y dependerá del método que llama al método **StartWorkflow** manejar la excepción y actuar en consecuencia.

public async Task MyMethodThatStartsAWorkflow()

{

try

{

await host.StartWorkflow("HelloWorld", 1, null);

}

catch(Exception ex) {

// Handle the exception appropriately

}

}

### Post Workflow Middleware

Este tipo de **middleware** se ejecutan una vez que se haya completado el **workflow** y se pueden usar para realizar acciones adicionales para todos los **workflow** de la aplicación.

En el ejemplo siguiente se muestra cómo puede utilizar un **middleware** de **workflow** **posterior** (**Post-Workflow**) para imprimir un resumen del **Workflow** en la consola.

Tenga en cuenta que utilice **WorkflowMiddlewarePhase.PostWorkflow** para especificar que se ejecuta una vez finalizado el **workflow**.

**Importante**: Debe llamar a siguiente como parte del middleware de flujo de trabajo para asegurarse de que se ejecuta el siguiente flujo de trabajo de la cadena.

public class PrintWorkflowSummaryMiddleware : IWorkflowMiddleware {

private readonly ILogger<PrintWorkflowSummaryMiddleware> \_log;

public PrintWorkflowSummaryMiddleware(

ILogger<PrintWorkflowSummaryMiddleware> log) {

\_log = log;

}

public WorkflowMiddlewarePhase Phase => WorkflowMiddlewarePhase.PostWorkflow;

public Task HandleAsync( WorkflowInstance workflow, WorkflowDelegate next) {

if (!workflow.CompleteTime.HasValue) { return next(); }

var duration = workflow.CompleteTime.Value - workflow.CreateTime;

\_log.LogInformation($@”Workflow {workflow.Description} completed in {duration:g}”);

foreach (var step in workflow.ExecutionPointers)

{

var stepName = step.StepName;

var stepDuration = (step.EndTime - step.StartTime) ?? TimeSpan.Zero;

\_log.LogInformation($“- Step {stepName} completed in {stepDuration:g}”);

}

return next();

}

}

### Exception Handling in Post Workflow Middleware

La gestión de errores en el **Post Workflow Middleware** se trata de manera diferente al **Pre Workflow middleware**.

En el momento en que se completa el **Workflow**, nuestro **Workflow** ya se ha ejecutado, por lo que es difícil actuar sobre una excepción no capturada.

Por defecto, si un **Middlewae workflow** lanza una excepción, será **logged** y el **workflow** se completará como si nada. Ese comportamiento se puede cambiar si se desea.

Para modificar el comportamiento por defecto de la gestión de excepción para todos los **Workflows** en nuestra app, registraremos una nueva implementación de la interface **IWorkflowMiddlewareErrorHandler** en la inyección de dependencias con el comportamiento personalizado que deseemos, tal y como se muestra a continuación:

// CustomMiddlewareErrorHandler.cs

public class CustomHandler : IWorkflowMiddlewareErrorHandler {

public Task HandleAsync(Exception ex) {

// Handle your error asynchronously

}

}

// Startup.cs

public void ConfigureServices(IServiceCollection services) {

// Other workflow configuration

services.AddWorkflow();

// Should go after .AddWorkflow()

services.AddTransient<IWorkflowMiddlewareErrorHandler, CustomHandler>();

}

## Registering Middleware

Para que el **middleware** surta efecto, deben registrarse en el gestor de inyección de dependencias incorporado utilizando los helpers o métodos adecuados.

**Nota:** El middleware se ejecutará en el orden en que se registran con middleware que se registran antes, ejecutándose antes en la cadena y terminando más tarde en la cadena. Para el middleware del Pre/Post **Workflow**, todo el middleware pre se ejecutará antes de que se inicie un flujo de trabajo y todo el middleware posterior se ejecutará después de que se complete un flujo de trabajo.

public class Startup {

public void ConfigureServices(IServiceCollection services) {

...

// Add workflow middleware

services.AddWorkflowMiddleware<AddDescriptionWorkflowMiddleware>();

services.AddWorkflowMiddleware<PrintWorkflowSummaryMiddleware>();

// Add step middleware

services.AddWorkflowStepMiddleware<LogCorrelationStepMiddleware>();

services.AddWorkflowStepMiddleware<PollyRetryMiddleware>();

...

}

}

## More Information

See the [Workflow Middleware](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample19) sample for full examples of workflow middleware in action.

# Multi-node clusters

By default, the **WorkflowHost** service will run as a single node using the built-in queue and locking providers for a single node configuration. Should you wish to run a multi-node cluster, you will need to configure an external queueing mechanism and a distributed lock manager to co-ordinate the cluster. These are the providers that are currently available.

De forma predeterminada, el servicio **WorkflowHost** se ejecutará como un solo nodo utilizando la cola integrada y los proveedores de bloqueo para una configuración de un solo nodo. Si desea ejecutar un clúster de varios nodos, deberá configurar un mecanismo de cola externo y un administrador de bloqueo distribuido para coordinar el clúster.

Estos son los proveedores que están disponibles actualmente.

## Queue Providers

* SingleNodeQueueProvider (Default built-in provider)
* [Azure Storage Queues](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.Azure)
* [Redis](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.Redis)
* [RabbitMQ](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.QueueProviders.RabbitMQ)
* [AWS Simple Queue Service](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.AWS)

## Distributed lock managers

* SingleNodeLockProvider (Default built-in provider)
* [Azure Storage Leases](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.Azure)
* [Redis](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.Redis)
* [AWS DynamoDB](https://github.com/danielgerlag/workflow-core/tree/master/src/providers/WorkflowCore.Providers.AWS)

# Using with ASP.NET Core

## How to configure within an ASP.NET Core application

En la clase **Startup**, use el método de extensión **AddWorkflow** para configurar los servicios principales del **workflow**, a continuación, registre los flujos de trabajo e inicie el host cuando configure la aplicación.

public class Startup {

public Startup(IConfiguration configuration) {

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services) {

services.AddMvc();

services.AddWorkflow(cfg => {

cfg.UseMongoDB(@”mongodb://mongo:27017”, “workflow”);

cfg.UseElasticsearch(new ConnectionSettings(new Uri(“http://elastic:9200”)), “workflows”);

});

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env) {

if (env.IsDevelopment()) {

app.UseDeveloperExceptionPage();

}

app.UseMvc();

var host = app.ApplicationServices.GetService<IWorkflowHost>();

host.RegisterWorkflow<TestWorkflow, MyDataClass>();

host.Start();

}

}

## Usage

Ahora simplemente inyecte los servicios que necesita en nuestros **controllers**

**IWorkflowController**

**IWorkflowHost**

**ISearchIndex**

**IPersistenceProvider**

# Elasticsearch plugin for Workflow Core

Es un **plugin** de “**search** **index**” para **Workflow Core** respaldado por **Elasticsearch,** que le permite indexar sus flujos de trabajo y buscar en los datos y el estado de los mismos.

## Installing

Install the NuGet package "**WorkflowCore.Providers.Elasticsearch**"

Using Nuget package console

PM> Install-Package WorkflowCore.Providers.Elasticsearch

Using .NET CLI

dotnet add package WorkflowCore.Providers.Elasticsearch

## Configuration

Use the .UseElasticsearch extension method on IServiceCollection when building your service provider

using Nest;

...

services.AddWorkflow(cfg =>

{

...

cfg.UseElasticsearch(new ConnectionSettings(new Uri("http://localhost:9200")), "index\_name");

});

## Usage

Inyecte el servicio **ISearchIndex** en el código y utilice el método **Search**.

Search(string terms, int skip, int take, params SearchFilter[] filters)

### terms

A whitespace separated string of search terms, an empty string will match everything. This will do a full text search on the following default fields

Una cadena de términos de búsqueda separada por espacios en blanco, una cadena vacía coincidirá con cualquier cosa. Esto hará una búsqueda de texto completo en los siguientes campos predeterminados

**Reference**

**Description**

**Status**

**Workflow Definition**

In **addition** you can search data within your own custom data object **if** it implements **ISearchable**

Además, puede buscar datos dentro de su propio objeto de datos personalizado si implementa **ISearchable**

using WorkflowCore.Interfaces;

...

public class MyData : ISearchable {

public string StrValue1 { get; set; }

public string StrValue2 { get; set; }

public IEnumerable<string> GetSearchTokens() {

return new List<string>() {

StrValue1,

StrValue2

};  
 }

}

#### Examples

Search all fields for "puppies"

searchIndex.Search("puppies", 0, 10);

#### skip & take

Use **skip** and **take** to page your search results. Where skip is the result number to start from and take is the page size.

#### filters

You can also supply a list of filters to apply to the search, these can be applied to both the standard fields as well as any field within your custom data objects. There is no need to implement **ISearchable** on your data object in order to use filters against it.

También puede proporcionar una lista de filtros que se puenden aplicar a la búsqueda, estos se pueden aplicar tanto a los campos estándar como a cualquier campo dentro de sus objetos de datos personalizados. No es necesario implementar **ISearchable** en su objeto de datos para usar filtros contra él.

Los siguientes tipos de filtro están disponibles

**ScalarFilter**

**DateRangeFilter**

**NumericRangeFilter**

**StatusFilter**

Estos existen en el namespace **WorkflowCore.Models.Search** .

#### Examples

Filtering by reference

using WorkflowCore.Models.Search;

...

searchIndex.Search("", 0, 10, ScalarFilter.Equals(x => x.Reference, "My Reference"));

**Filtering** by **workflows** started after a date

searchIndex.Search("", 0, 10, DateRangeFilter.After(x => x.CreateTime, startDate));

**Filtering** by **workflows** completed within a period

searchIndex.Search("", 0, 10, DateRangeFilter.Between(x => x.CompleteTime, startDate, endDate));

**Filtering** by workflows in a state

searchIndex.Search("", 0, 10, StatusFilter.Equals(WorkflowStatus.Complete));

**Filtering** against your own custom data class

class MyData

{

public string Value1 { get; set; }

public int Value2 { get; set; }

}

searchIndex.Search("", 0, 10, ScalarFilter.Equals<MyData>(x => x.Value1, "blue moon"));

searchIndex.Search("", 0, 10, NumericRangeFilter.LessThan<MyData>(x => x.Value2, 5))

# Test helpers for Workflow Core

Proporciona soporte para pruebas de escritura para flujos de trabajo basados en **WorkflowCore**

## Installing

Install the NuGet package "**WorkflowCore.Testing**"

PM> Install-Package WorkflowCore.Testing

### Usage

### With xUnit

Create a class that inherits from **WorkflowTest**

Call the **Setup**() method in the constructor

Implemente sus pruebas utilizando los métodos auxiliares

**StartWorkflow()**

**WaitForWorkflowToComplete()**

**WaitForEventSubscription()**

**GetStatus()**

**GetData()**

**UnhandledStepErrors**

public class xUnitTest : WorkflowTest<MyWorkflow, MyDataClass> {

public xUnitTest() { Setup(); }

[Fact]

public void MyWorkflow()

{

var workflowId = StartWorkflow(new MyDataClass() { Value1 = 2, Value2 = 3 });

WaitForWorkflowToComplete(workflowId, TimeSpan.FromSeconds(30));

GetStatus(workflowId).Should().Be(WorkflowStatus.Complete);

UnhandledStepErrors.Count.Should().Be(0);

GetData(workflowId).Value3.Should().Be(5);

}

}

### With NUnit

* **Create** a **class** that inherits from **WorkflowTest** and decorate it with the ***TestFixture*** attribute
* **Override** the **Setup** method and decorate it with the ***SetUp*** attribute

Implemente sus pruebas utilizando los métodos auxiliares

**StartWorkflow()**

**WaitForWorkflowToComplete()**

**WaitForEventSubscription()**

**GetStatus()**

**GetData()**

**UnhandledStepErrors**

[TestFixture]

public class NUnitTest : WorkflowTest<MyWorkflow, MyDataClass> {

[SetUp]

protected override void Setup() { base.Setup(); }

[Test]

public void NUnit\_workflow\_test\_sample() {

var workflowId = StartWorkflow(new MyDataClass() { Value1 = 2, Value2 = 3 });

WaitForWorkflowToComplete(workflowId, TimeSpan.FromSeconds(30));

GetStatus(workflowId).Should().Be(WorkflowStatus.Complete);

UnhandledStepErrors.Count.Should().Be(0);

GetData(workflowId).Value3.Should().Be(5);

}

}

## Extensions

* [User (human) workflows](https://github.com/danielgerlag/workflow-core/tree/master/src/extensions/WorkflowCore.Users)

https://github.com/danielgerlag/workflow-core/tree/master/src/extensions/WorkflowCore.Users

# Samples

|  |  |
| --- | --- |
| [Hello World](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample01) | <https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample01> |
| [Passing Data](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample03) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample03 |
| [Events](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample04) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample04 |
| [Activity Workers](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample18) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample18 |
| [Dependency Injection](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample15) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample15 |
| [Parallel ForEach](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample09) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample09 |
| [While loop](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample10) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample10 |
| [If](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample11) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample11 |
| [Parallel Tasks](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample13) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample13 |
| [Saga Transactions](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample17) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample17 |
| [Scheduled Background Tasks](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample16) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample16 |
| [Recurring Background Tasks](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample14) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample14 |
| [Multiple outcomes](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample12) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample12 |
| [Deferred execution & re-entrant steps](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample05) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample05 |
| [Looping](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample02) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample02 |
| [Exposing a REST API](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WebApiSample) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WebApiSample |
| [Human(User) Workflow](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample08) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample08 |
| [Workflow Middleware](https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample19) | https://github.com/danielgerlag/workflow-core/tree/master/src/samples/WorkflowCore.Sample19 |