PineCone.net

Introduction

PineCone.net is a cross-platform workflow\integration framework based on .NET Core. The framework provides a platform to build complex systems that can integrate with different systems across different platforms.

The framework supports long running workflows which include synchronous as well as asynchronous tasks.

Since, the framework is written in .NET Core, it can be deployed on different Operating Systems. One easy way to package the different components is by creating Docker containers for each component. Hence, the framework can be run both as an on-premise application as well as a cloud application or the combination of the two.

The framework supports writing your own engine providers. Some of the current integration systems like BizTalk (which is not supported in .NET Core) use SQL Server as the engine.

However, PineCone.net can be hooked onto different engines. The framework provides support to build your own engines.

At the moment the framework has built in support for the AWS (Amazon Web Services) and MySql engines.

The framework supports JSON data format which is less bulkier and easier to read than XML.

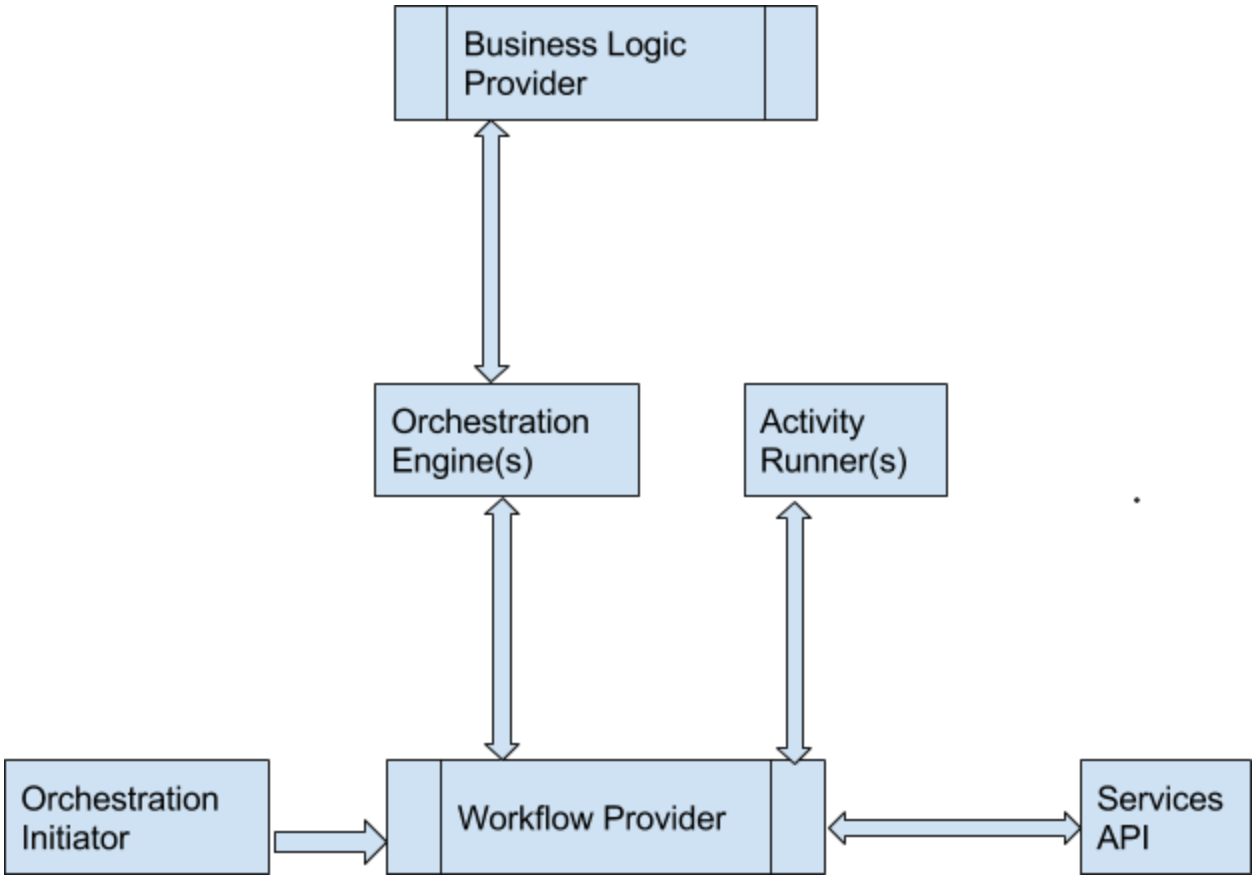
AWS Simple Workflow Service(SWF) provides a very simple and versatile backend system for orchestration of workflows. However, SWF requires you to develop your own ‘deciders’ which orchestrate different tasks. This can be a pain sometimes. PineCone.net provides an orchestration engine which hooked into SWF when AWS is chosen as the backend engine. The business logic for the orchestration is stored in the DynamoDB (NoSQL database in AWS).

The business logic is a simple JSON file which is described later in this document and can be easily uploaded via. A REST interface provided by the framework.

System Architecture

System Components

The following diagram illustrates the different components of the framework at play.



Orchestration Engine

The Orchestration Engine is the central component of the system. This is the component that makes decisions on workflows based on the workflow states and business logic configuration.

An Orchestration Engine is assigned for every host process. A host process can host one or more workflows.

For very large systems only one workflow must be hosted by a host process. For medium range systems, it is ideal to host all related workflows in a single host process while in small systems all workflows can be hosted on the same process.

However, the hosting decisions should be made only after considering the size, volume and cost of running.

These are the sequence of tasks carried out by the Orchestration Engine:-

1. Read the current workflow state.
2. Read the business logic configuration for that workflow.
3. Make decisions (like scheduling activity, failing workflow, completing workflow) based on the workflow state and business logic.

Orchestration Initiator

The Orchestration Initiator is the component that initiates the main workflow process. An initiator calls an interface method in the Workflow provider interface to initiate the workflow. This is covered in the next section (Using the PineCone framework).

An example of simple initiator would be a component that reads from Queue or calls a REST Service to get data and initiates a workflow.

PineCone.net provides a comprehensive framework to write such a framework with minimal effort. However, in the future standard configurable initiators shall be made available to read from different systems, just like you have one-way incoming ports to start an orchestration in BizTalk.

Activity Runner

If you consider the Initiator as a one-way incoming port to start an orchestration, Activity Runner can be considered a two-way outgoing-incoming port inside the orchestration process.

Activity Runner supports both synchronous as well as asynchronous tasks that require human input.

Just like the Orchestration Engine, Activity Runner runs on a host process and the choice of hosting single or multiple activity runners on a host should be made after considering the volume and cost of the system.

These are the tasks carried out by the Activity Runner:-

1. Get the context of the activity from the workflow.
2. Execute a synchronous task and notify the workflow of completion or failure of task. An example of this task would be calling a REST service and returning the response to the workflow.

3. Execute an asynchronous outgoing task. An example of this task would be sending an email or writing data to a queue.
4. Receive response from another system and notify the workflow of completion or failure of task. An example of this task would be reading from a queue or reading from a database table until a value is found and returning the response to the workflow.

Just like the initiator, writing an activity runner is very simple as the framework provides an interface that can be implemented to write an activity runner. However, standard configurable activity runners shall be made available a part of the framework in the future.

Service API

The PineCone.net framework is not shipped with a fancy management management console like the BizTalk management console. However, the framework has in built REST API(s) which accept JSON inputs for workflow operation and management.

Workflows themselves are represented using JSON data. Hence, workflows can be easily created by using the Business Logic API.

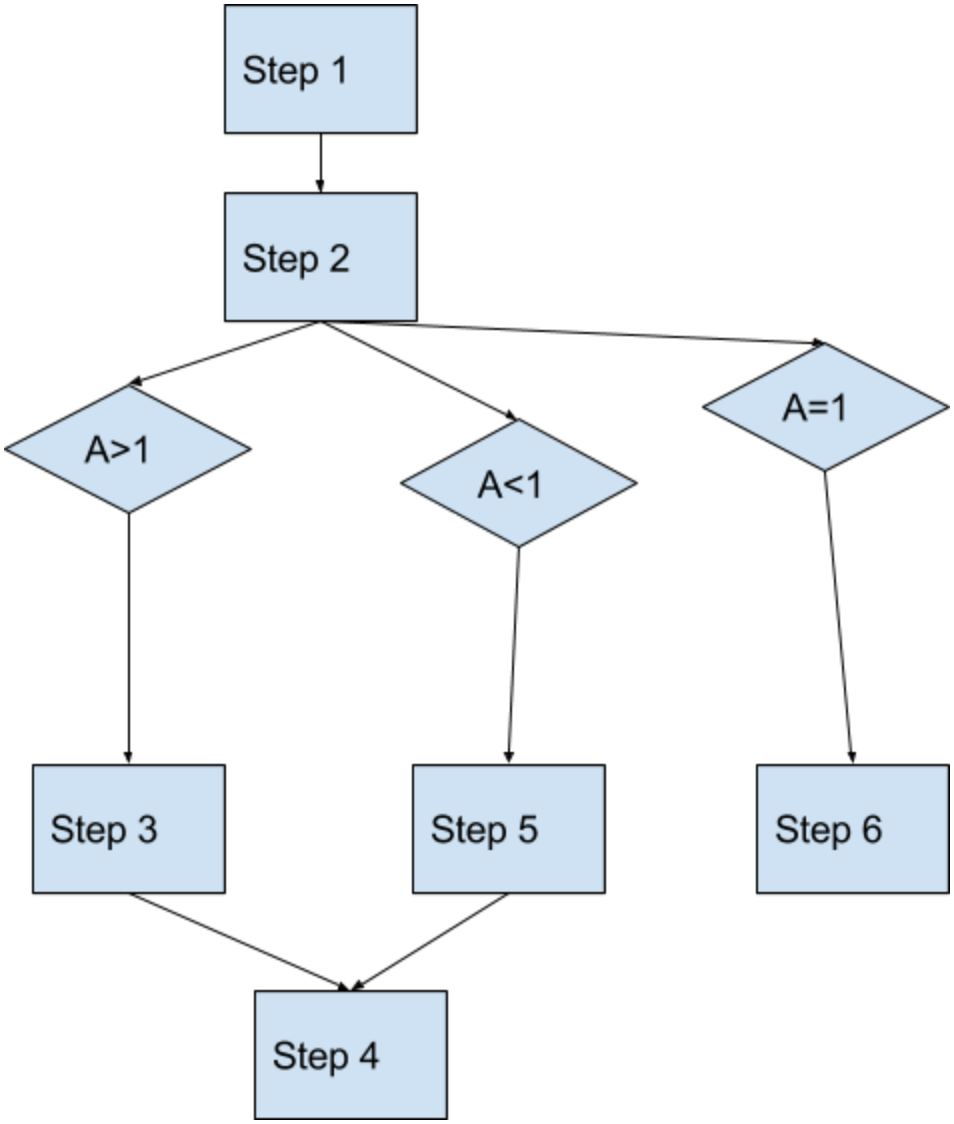
Workflows can be managed using the Management API.

The API gives the flexibility to the UI developers to develop their own fancy management consoles on different platforms.

Creating Workflows

Simple Workflow

A simple workflow is set of sequential steps. A simple workflow can also have conditional branches. The diagram below illustrates a simple workflow.



The above workflow is a simple workflow which consists of steps and for each of three conditions there is only one sequence of steps that can occur.
There are three possibilities in this case.

1. When A>1, we have step 1,2,3 & 4.
2. When A<1, we have step 1,2,5 & 4.
3. When A = 1, we have step 1,2 & 6.

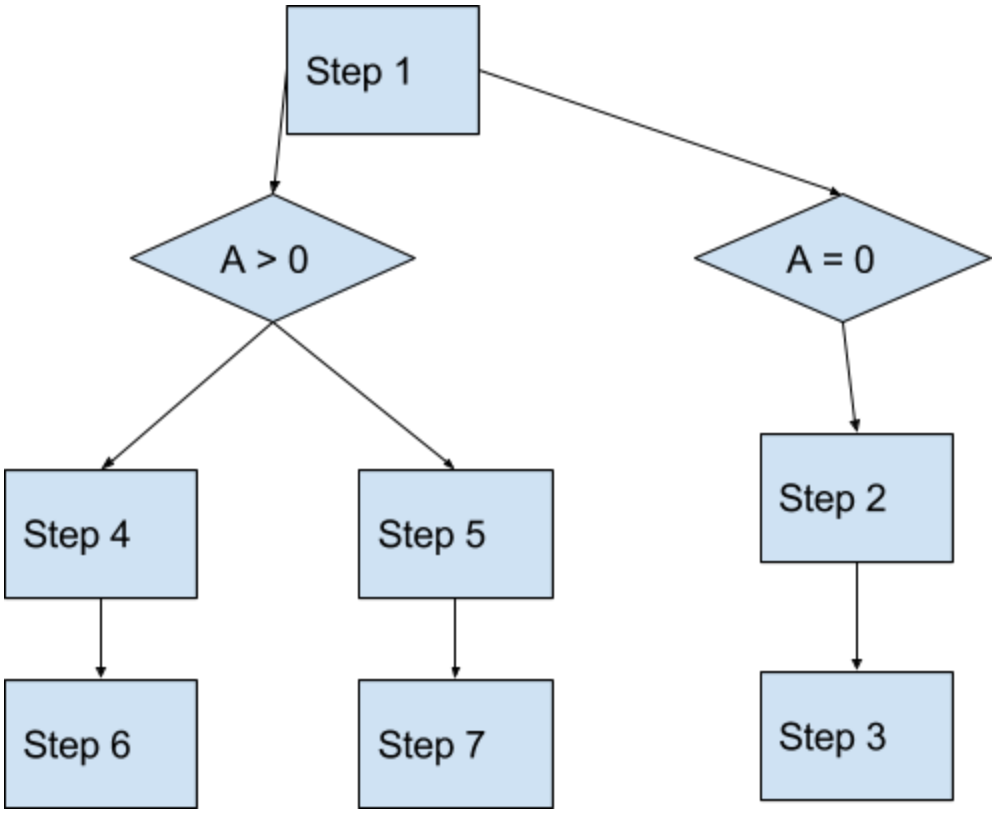
This is a correct design for a simple workflow.

Child Workflow

In the last section we learnt how to design a simple workflow. But what if we want parallel branches where there can be more than one branch for a condition?

Consider the scenario below. When we have A = 0, the workflow is pretty straight forward i.e. Step 1, 2 & 3. While on the other hand when A > 0, we want the workflow to run two parallel flows. However, the way the workflow has been designed below we cannot predict what will happen in case of A >0 as the workflow will finish as soon as the last step (Step 6 or 7) is executed.

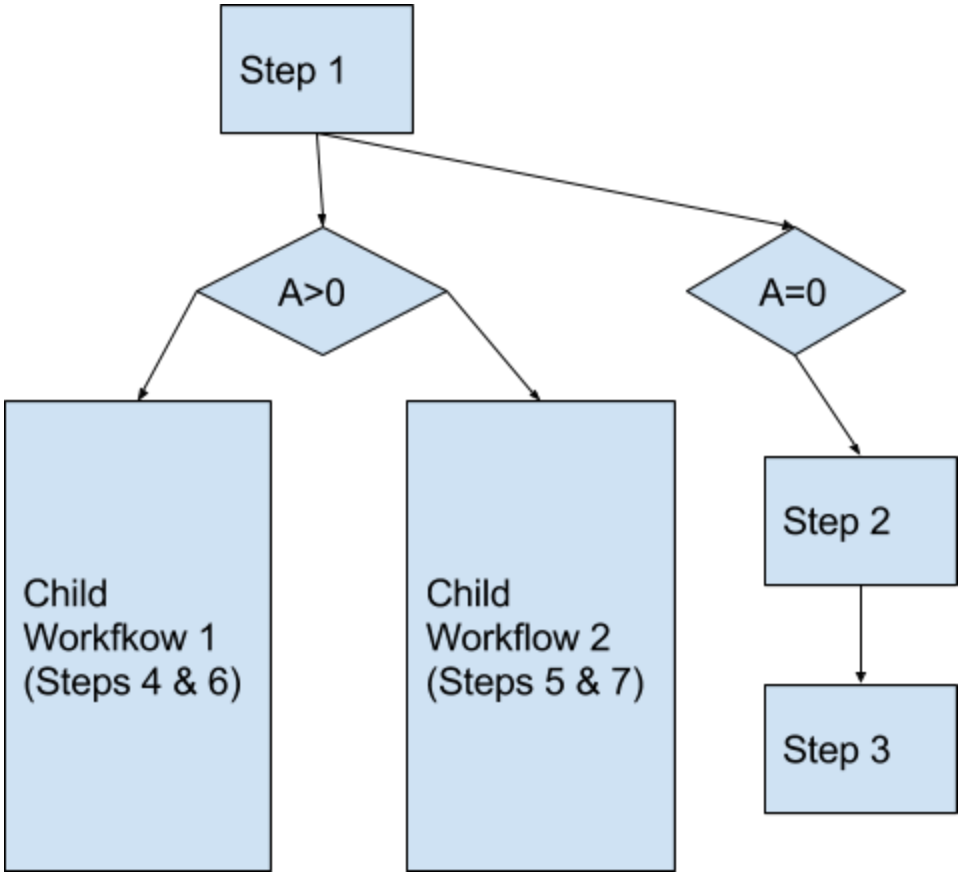
.



Therefore, we have the following 5 possibilities in case of A>0

1. Step 4 & 6.
2. Step 5 & 7.
3. Step 4,6, & 5.
4. Step 5,7 & 4.
5. Step 4,5,6 & 7.

Which of the above possibilities occurs, depends upon the time consumed by each step. To solve such a complex scenario we need to include child workflows inside the main workflow. The correct design for this scenario is illustrated below.



With the above design in case of A >0, we have the two parallel branches executed as two independent child workflows and hence we are guaranteed of running all the desired steps (4,5,6 &7).

Conditions

Whenever we need to branch inside a workflow we need to add a condition to the first step of that branch.

A condition simple consists of an **LHS** (Left Hand Side) expression , a **RHS** (Right Hand Side) Expression and a Operator. The expression can either be value or a JSON path expression.

These are the operators supported at the moment by the framework:-

1. Strigequals.
2. Stringcontains
3. Mathequals
4. Mathgreaterthan
5. Mathlessthan
6. Mathgreaterthanorequalto
7. Mathlessthanorequalto

Example of a condition:

```
"Lhs": {
                "JsonExpression":
"$.Data.Branch[?(@.SubBranch.Leaf=='Green')].Flower"
        },
"Operator": "StringEquals",
 "Rhs": {
                "JsonValue": "Red”
         }
```

For the following JSON the above expression is evaluated to **TRUE**:

```
{ “Data”:
        { “Branch” :
                {“SubBranch” :
                        {“Leaf”:”Green”},
                “Flower”:”Red”
                }
        }
}
```

A step can have several condition groups, with each condition group consisting of several conditions.

A step is executed when any one of the condition groups is evaluated to TRUE. A condition group is evaluated to true when all its conditions are evaluated to true.

For example a step contains the 3 condition groups:

1. Group 1 consists of 2 conditions , Country = Norway and Season = Winter.
2. Group 2, Country = Spain and Season = Summer.
3. Group 3, Country = Australia and Season = Summer.

The step would be run in case of any of three condition groups (each one of them consists of two conditions) is satisfied. Each of the three condition groups is satisfied when both the conditions inside the group are satisfied.

Workflow Policies

A workflow policy determines what happens to the workflow in case of step completion or failure.

A task policy consists of :-

1. Retry count (how many times a task should be retried in case of failure).
2. Action (whether to stop or continue the workflow).

A workflow has two task policies:-

1. **TaskSuccessPolicy** (typically this would be set to Action=Continue), no retry count.
2. **TaskFailurePolicy** (typically this would have a retry count and would be set to Action=Stop.However, in some cases it can also be set to Action=Continue with retry count).

Business Logic Workflow JSON

A workflow is represented using a JSON object. One JSON object contains a collection of workflows which are hosted by the same host. So, we have one JSON object per host.

Each workflow in turn contains the workflow policies and steps which has it's own condition groups as described in the above section. A step either be an **Activity** or it can be a **Child Workflow**.

An activity step is represented by:-

1. **ActivityID** : The reference ID for the step in the workflow.
2. **ActivityName** : The name of the activity in the workflow engine (ActivityTypeName in AWS SWF).
3. **Version**: The version of the activity.
4. **PreviousStepID**: The reference to the previous step. NULL if the first step.

Similarly, a child workflow step is represented by:-

1. **WorkflowID**: The reference ID for the step.
2. **WorkflowTypeName**: The name of the child workflow type in the workflow engine.
3. **Version**: the version of the workflow.
4. **PreviousStepID**: The reference to the previous step.

Below is the JSON that we would use for the complex workflow scenario that we designed above in the Child Workflows section.

It is upto the individual BusinessLogic providers to decide the format of the JSON(s). Hence, no JSON schema is created yet. The JSON below is used by DynamoDB and MySql providers.

```
{
  "HostName": "RootHost",
  "Workflows": [
        {
        "WorkflowTypeName": "MainWorkflow",
        "TaskSuccessPolicy": {
        "WorkflowAction": "Continue"
        },
        "TaskFailurePolicy": {
        "RetryCount": "5",
        "WorkflowAction": "Stop"
        },
        "Steps": [
        {
        "ActivityID": "Step1",
        "ActivityName": "SendMessageToQueue",
        "Version": "1.0"
        },
        {
        "ActivityID": "Step2",
        "ActivityName": "SendReport",
        "PreviousStepID": "Step1",
        "Version": "1.0",
        "ConditionGroups": [
        {
        "Conditions": [
                {
                "Lhs": {
                "JsonExpression": "$.Data.A"
                },
                "Operator": "Mathequals",
                "Rhs": {
                "JsonExpression": "0"
                }
                }
        ]
        }
```

```
]
},
{
"WorkflowID": "ChildWorkflow1",
"WorkflowTypeName": "ImageProcessing",
"PreviousStepID": "Step1",
"Version": "1.0",
"ConditionGroups": [
{
"Conditions": [
        {
        "Lhs": {
        "JsonExpression": "$.Data.A"
        },
        "Operator": "Mathegreaterthan",
        "Rhs": {
        "JsonExpression": "1"
        }
        }
]
}
]
},
{
"WorkflowID": "ChildWorkflow2",
"WorkflowTypeName": "WordProcessing",
"PreviousStepID": "Step1",
"Version": "1.0",
"ConditionGroups": [
{
"Conditions": [
        {
        "Lhs": {
        "JsonExpression": "$.Data.A"
        },
        "Operator": "Mathegreaterthan",
        "Rhs": {
        "JsonExpression": "1"
        }
        }
]
}
]
```

```
}
]
},
{
"WorkflowTypeName": "ImageProcessing",
"TaskSuccessPolicy": {
"WorkflowAction": "Continue"
},
"TaskFailurePolicy": {
"RetryCount": "5",
"WorkflowAction": "Stop"
},
"Steps": [
{
"ActivityID": "Step4",
"ActivityName": "MakeImage",
"Version": "1.0"
},
{
"ActivityID": "Step6",
"ActivityName": "PublishImage",
"Version": "1.0",
"PreviousStepID": "Step4"
}
]
},
{
"WorkflowTypeName": "WordProcessing",
"TaskSuccessPolicy": {
"WorkflowAction": "Continue"
},
"TaskFailurePolicy": {
"RetryCount": "5",
"WorkflowAction": "Stop"
},
"Steps": [
{
"ActivityID": "Step5",
"ActivityName": "CreateDictionay",
"Version": "1.0"
},
{
"ActivityID": "Step7",
```

```
        "ActivityName": "AddSpecialCharacters",
        "Version": "1.0",
        "PreviousStepID": "Step5"
        }
        ]
        }
 ]
}
```

Using the PineCone framework

Creating an Activity Runner

Activity runner is the worker process that connects to external systems and carries out workflow tasks. It is extremely easy to create an activity runner. We just need to configure it correctly and implement a worker.

Configuring an Activity Runner

An activity runner is just a console application which can either be run on a host machine using a **'dotnet run**' command or it can also be run as a docker container using the **docker run** command. However, we need to configure it before we can run it.

The **appsettings.json** contains the configuration:-

1. WorkflowProvider: This is configured to the provider we want to use. At the moment we have the AWS SWF and the MySql providers available. This is loaded using reflection.
2. ActivityRunner: We need to specify the assembly of the worker since we load it using reflection too.
3. WorkerType: This is configured to one of the three values-SyncRun,AsycOutGoing or AsyncIncommingPoll.
4. HostName: The name of the host process. Task list for AWS SWF.
5. PollingIntervalInMilliseconds: The time interval between two polls.

The following is an example of an activity runner configuration using the MySql workflow provider:-

```
{
  "WorkflowProvider": {
        "Assembly": {
        "AssemblyName": "PineCone.WorkflowProvider.MySql",
        "TypeName": "PineCone.WorkflowProvider.MySql.MySqlProvider",
        "Version": "1.0.0.0"
```

```
        }
 },
  "ActivityRunner": {
        "Assembly": {
        "AssemblyName": "PineCone.SimpleSync",
        "TypeName": "PineCone.SimpleSync.SyncRunner",
        "Version": "1.0.0.0"
        },
        "WorkerType": "SyncRun"
 },
 "HostName": "TestList",
 "PollingIntervalInMilliseconds": "10"
}
```

The following is an example of an activity runner configuration using the AWS SWF provider:-

```
{
 "WorkflowProvider": {
   "Assembly": {
        "AssemblyName": "PineCone.WorkflowProvider.AWS",
        "TypeName": "PineCone.WorkflowProvider.AWS.AWSWorkflowProvider",
        "Version": "1.0.0.0"
   }
 },
 "ActivityRunner": {
        "Assembly": {
        "AssemblyName": "PineCone.SimpleSync",
        "TypeName": "PineCone.SimpleSync.SyncRunner",
        "Version": "1.0.0.0"
        },
        "WorkerType": "SyncRun"
 },
 "HostName": "TestList",
 "PollingIntervalInMilliseconds": "10"
}
```

Implementing a worker

To implement a worker, we just need to create a simple .NET Core class library and add a reference to the assembly - **"PineCone.Model": "1.0.0-*".**

We just need to derive from the base class **PineCone.Model.Activity**.**ActivityWorkerBase** and override one of the following 3 methods depending upon the worker type:-

1. protected virtual ActivityInstance SyncRun(ActivityInstance instance).
2. protected virtual void AsycOutGoing(ActivityInstance instance).
3. protected virtual ActivityInstance AsyncIncommingPoll().

Below is an example of a SyncRun that just adds some data to the existing instance:-

```
using PineCone.Model.Activity;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Newtonsoft.Json;
namespace PineCone.SimpleSync
{
        public class SyncRunner : ActivityWorkerBase
        {
        protected override ActivityInstance SyncRun(ActivityInstance instance)
        {
        instance.Data = JsonConvert.SerializeObject(
                new { Runner = "DemoSample", Data = instance.Data, UniqueID =
Guid.NewGuid().ToString() });
        instance.Status = Model.Activity.TaskStatus.SUCCESS;
        return instance;
        }
        }
}
```

We can write code to call a REST service, read data from a database or queue, read content from a file or an email and use that data in the activity instance..

In the future such out-of-the-box activity runners shall be made available as part of the framework.

Creating an Orchestration Initiator

The orchestration initiator is also a .NET Core console application which runs continuously and tries to initiate a workflow.

Configuring the initiator process

Just like the activity runner, the initiator can either be run by dotnet run or by docker. We configure it the same way in the appsettings.

The **appsettings.json** contains the configuration:-

1. WorkflowProvider: This is configured to the provider we want to use. At the moment we have the AWS SWF and the MySql providers available. This is loaded using reflection.
2. OrchestrationInitiator: We need to specify the assembly of the initiator since we load it using reflection too.
3. Domain: The name of the domain of the workflow.
4. WorkflowTypeName: The name of the workflow type.
5. Version: The version of the workflow type.
6. PollingIntervalInMilliseconds: The time interval between two polls.

The following is an example of an initiator using the AWS workflow provider:-

```
{
  "WorkflowProvider": {
    "Assembly": {
          "AssemblyName": "PineCone.WorkflowProvider.AWS",
          "TypeName": "PineCone.WorkflowProvider.AWS.AWSWorkflowProvider",
          "Version": "1.0.0.0"
    },
          "OrchestrationInitiator": {
          "Assembly": {
          "AssemblyName": "PineCone.DemoInitiator",
          "TypeName": "PineCone.DemoInitiator.Initiator",
          "Version": "1.0.0.0"
          }
          },
          "PollingIntervalInMilliseconds": "10000",
          "Domain": "PineCone",
          "WorkflowTypeName": "Test",
          "Version": "1.0"

  }
```

The following is an example of an initiator using the MySql workflow provider:-

```
{
          "WorkflowProvider": {
          "Assembly": {
          "AssemblyName": "PineCone.WorkflowProvider.MySql",
```

```
        "TypeName": "PineCone.WorkflowProvider.MySql.MySqlProvider",
        "Version": "1.0.0.0"
        }
        },
        "OrchestrationInitiator": {
        "Assembly": {
        "AssemblyName": "PineCone.DemoInitiator",
        "TypeName": "PineCone.DemoInitiator.Initiator",
        "Version": "1.0.0.0"
        }
        },
        "PollingIntervalInMilliseconds": "10000",
        "Domain": "PineCone",
        "WorkflowTypeName": "Test",
        "Version": "1.0"

}
```

Implementing an initiator

To implement an initiator, we just need to create a simple .NET Core class library and add a reference to the assembly - **"PineCone.Model": "1.0.0-*".**

We need to implement the interface **PineCone.Model.Orchestration.IOrchestrationInitiator** and implement the following method:-
void InitiateWorkflow(string workflowTypeName, string version, string domain, IWorkflowProvider provider).

Below is an example of a simple initiator:-

```
using PineCone.Model.Orchestration;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using PineCone.Model;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
namespace PineCone.DemoInitiator
{
        public class Initiator : IOrchestrationInitiator
        {
```

```
        public void InitiateWorkflow(string workflowTypeName, string version, string domain,
IWorkflowProvider provider)
        {


        provider.InitiateWorkflow(workflowTypeName, version, domain,
                JsonConvert.SerializeObject(new
                {   InitiatedDateTime = DateTime.UtcNow.ToString("yyyyMMdd:HHmmss"),
                ID = Guid.NewGuid().ToString(),
                Data =  "zyz"
                }));
        }
        }
}
```

The above sample of course is the simplest example one can image. We can write code to call a REST service, read data from a database or queue, read content from a file or an email and use that data to initiate the orchestration.

In the future such out-of-the-box initiators shall be made available as part of the framework.

Configuring the Orchestration Engine

The orchestration engine is the central process which controls the workflow. It interacts with both the business logic provider and the workflow provider. It is also a .NET Core applications which runs continuously and can be run using the dotnet run or in docker.

This is how we need to configure the appsettings:-

1. WorkflowProvider: This is configured to the provider we want to use. At the moment we have the AWS SWF and the MySql providers available. This is loaded using reflection.
2. BusinessLogicProvider: Just like the above. We have the AWS DynamoDB and and MySql available at the moment.
3. HostName: The name of the host process. Task list for AWS SWF.
4. PollingIntervalInMilliseconds: The time interval between two polls.

Example configuration using the MySql providers:-

```
{
  "BusinessLogicProvider": {
        "Assembly": {
        "AssemblyName": "PineCone.BusinessLogicProvider.MySql",
        "TypeName": "PineCone.BusinessLogicProvider.MySql.MySqlProvider",
        "Version": "1.0.0.0"
```

```
        }
 },
 "WorkflowProvider": {
        "Assembly": {
        "AssemblyName": "PineCone.WorkflowProvider.MySql",
        "TypeName": "PineCone.WorkflowProvider.MySql.MySqlProvider",
        "Version": "1.0.0.0"
        }
 },
 "HostName": "TestList",
 "PollingIntervalInMilliseconds": "10"
}
```

Example configuration using the AWS providers:-

```
{
 "BusinessLogicProvider": {
   "Assembly": {
        "AssemblyName": "PineCone.BusinessLogicProvider.DynamoDB",
        "TypeName": "PineCone.BusinessLogicProvider.DynamoDB.DynamoProvider",
        "Version": "1.0.0.0"
         }
 },
 "WorkflowProvider": {
   "Assembly": {
        "AssemblyName": "PineCone.WorkflowProvider.AWS",
        "TypeName": "PineCone.WorkflowProvider.AWS.AWSWorkflowProvider",
        "Version": "1.0.0.0"
   }
 },
 "HostName": "TestList",
 "PollingIntervalInMilliseconds": "10"
}
```

Configuring Providers

This section briefly describes how we can configure the 4 providers which are available to us today.

AWS Workflow Provider

This uses the AWS SWF(Simple Workflow) as the backend engine. To configure this as the workflow provider first we need to add the WorkflowProvider section in the appsettings of the

Activity Runner, Orchestration Initiator or the Orchestration Engine as described in the earlier sections.

In addition, we also need to add **AWSWorkflowProvider.json** to the Activity Runner, Orchestration Initiator or the Orchestration Engine directory. This contains the AWS specific configuration for your environment.

This file must contain the following:-

```
{
  "Region": "us-west-2",
  "AccessKeyID": "xxx",
  "SecretAccessKey": "zzz",
  "User": "swf-user",
  "SWFDomain": "PineCone"
}
```

DynamoDB BusinessLogic Provider

This uses the DynamoDB NoSQL database as the business logic provider.

Just like the SWF provider, in addition to adding to appsettings, we need to add the **DynamoDBBusinessLogicProvider.json** file.

The file must contain the following configuration:-

```
{
  "Region": "us-west-2",
  "AccessKeyID": "xxx",
  "SecretAccessKey": "zzz",
  "User": "swf-user",
  "TableName": "PineCone_Orchestration"
}
```

The table “PineCone_Orchestration” has the following 2 keys:
Partition Key: HostName (string)
Sort key:WorkflowTypeName (string)

MySql Workflow Provider

This uses MySql database as the workflow provider. In addition to the appsettings, we must add the file **MySqlWorkflowProvider.json**, containing the connection string to the MySql database:-

```
{
```

```
  "ConnectionString":
"Server=localhost;Port=3306;Database=pinecone_businesslogic;Uid=root;Pwd=xxx"
}
```

MySql BusinessLogic Provider

This uses MySql database as the business logic provider. In addition to the appsettings, we must add the file **MySqlBusinessLogicProvider.json**, containing the connection string to the MySql database:-

```
{
  "ConnectionString":
"Server=localhost;Port=3306;Database=pinecone_businesslogic;Uid=root;Pwd=xxx"
}
```

Using the Services API

The Services API provides a REST interface to operate and manage the workflows and the business logic around the workflows.

The Services API is a .NET Core Web API application which runs on the **Kestrel** web server. Hence, this too can either be run locally or inside a docker container.

Configuring the Services API

Just like the Orchestration Engine, the Services API is configured by specifying the BusinessLogic and Workflow providers in the appsettings. In addition, we also need to specify the Management provider in the appsettings.

Below is an example of appsettings which uses MySql as all 3 providers:-

```
{
"BusinessLogicProvider": {
        "Assembly": {
        "AssemblyName": "PineCone.BusinessLogicProvider.MySql",
        "TypeName": "PineCone.BusinessLogicProvider.MySql.MySqlProvider",
        "Version": "1.0.0.0"
        }
  },
  "ManagementProvider": {
        "Assembly": {
        "AssemblyName": "PineCone.WorkflowProvider.MySql",
```

```
        "TypeName": "PineCone.WorkflowProvider.MySql.MySqlManagementProvider",
        "Version": "1.0.0.0"
        }
 },
  "WorkflowProvider": {
        "Assembly": {
        "AssemblyName": "PineCone.WorkflowProvider.MySql",
        "TypeName": "PineCone.WorkflowProvider.MySql.MySqlProvider",
        "Version": "1.0.0.0"
        }
 }
}
```

We also need to add the **MySqlWorkflowManagementProvider.json** which contains the MySql connection string.

```
{
  "ConnectionString":
"Server=localhost;Port=3306;Database=pinecone_businesslogic;Uid=root;Pwd=xxx"
}
```

Business Logic API

The business logic API is used to manage the business logic around the workflows. This API is available for both the DynamoDB as well as the MySql business logic providers.

The following methods are provided by this API-

GET api/BusinessLogic/{hostName}/{workflowTypeName}
Returns : 200 OK with the business logic workflow JSON object.

POST api/BusinessLogic/{hostName}
Body : Business logic workflow JSON object.
Returns : 204 No Content. Adds the workflow JSON to the business logic provider.

PUT api/BusinessLogic/{hostName}
Body : Business logic workflow JSON object.
Returns : 204 No Content. Replaces the current workflow JSON with this new one in the business logic provider .

DELETE api/BusinessLogic/{hostName}/{workflowTypeName}
Returns : 204 No Content. Deletes the workflow JSON object from the business logic provider.

Workflow API

This API provides two basic workflow operation methods. This API is available for both AWS SWF provider as well as the MySql workflow provider.

Microsoft BizTalk provides a tool to expose an orchestration as either a Web Service or a WCF Service. This API provides a somewhat similar feature.

The workflow API provides two methods in by which the orchestration can be initiated or run using REST calls. We do not need to expose the orchestration as a REST service. This feature comes out-of-the-box with this API.

POST api/Workflow/Initiate/{domainName}/{workflowTypeName}/{version}
Returns : 204 No Content. Initiates the workflow.

POST api/Workflow/RunSynchronous/{domainName}/{workflowTypeName}/{version}
Returns : 200 Ok.Runs the workflow synchronously and returns the result of the workflow execution. In case of MySql provider, the result of the last executed activity.

Workflow Management API

This API provides methods to manage the workflows. This is only available for the MySql provider as the AWS console comes with a management console for SWF.

GET api/WorkflowManagement/Domains
Returns: 200 Ok with the list of configured domains.

GET api/WorkflowManagement/Domain/{domainName}/ActivityTypes
Returns: 200 Ok with the list of activity types for that domain.

GET api/WorkflowManagement/Domain/{domainName}/WorkflowTypes
Returns: 200 Ok with the list of workflow types for that domain.

POST api/WorkflowManagement/Domain/{domainName}
Returns: 204 No Content.Adds the domain to the workflow provider.

DELETE api/WorkflowManagement/Domain/{domainName}
Returns: 204 No Content.Deprecates the domain.

POST api/WorkflowManagement/WorkflowType/{workflowTypeName}/{version}
Body : {
"HostName": {hostname},
"Timeout": {timeoutinseconds},
"DecisionTimeout": {decisiontimeoutinseconds},
"DomainName": {domainname}
}
Returns: 204 No Content. Adds the workflow type to the workflow provider.

DELETE api/WorkflowManagement/Domain/{domainName}/WorkflowType/{workflowTypeName}/{version}
Returns: 204 No Content. Deprecates the workflow type.

POST api/WorkflowManagement/ActivityType/{activityTypeName}/{version}
Body :{
"HostName": {hostname},
"Timeout": {timeoutinseconds},
"DomainName": {domainname}
}
Returns: 204 No Content. Adds the activity type to the workflow provider.

DELETE api/WorkflowManagement/Domain/{domainName}/ActivityType/{activityTypeName}/{version}
Returns: 204 No Content. Deprecates the activity type.

DELETE api/WorkflowManagement/Workflow/{workflowExecutionID}
Returns: 204 No Content. Terminates the workflow execution.

GET api/WorkflowManagement/Domain/{domainName}/Workflows
Returns: 200 Ok with the list of workflow executions.

GET api/WorkflowManagement/Domain/Workflow/{workflowExecutionID}/Activities
Returns: 200 Ok with the list of workflow execution activities.

GET api/WorkflowManagement/Workflow/{workflowExecutionID}/Events
Returns: 200 Ok with the list of workflow execution events.

Creating your own engine

The PineCone.net framework comes with Business Logic providers for AWS DynamoDB database and MySql database and Workflow providers for AWS SWF and MySql. However, in the future it is planned to build several providers for other data sources like Microsoft SQL Server, Oracle, MongoDB, DocumentDB and Cassandra.

It is also possible to build your own providers on top of your favorite data source. A data source can be something as simple as an XML file or a JSON file or just a delimited flat file.

The PineCone.net framework provides interfaces to build the providers.

Creating your own BusinessLogic provider

To implement your own BusinessLogic provider, we just need to create a simple .NET Core class library and add a reference to the assembly - **"PineCone.Model": "1.0.0-*".**

We need to implement the interface **PineCone.Model.IBusinessLogicProvider** and implement the following methods:-

1. TaskPolicy GetTaskSuccessPolicy(string workflowTypeName,string hostName).
2. TaskPolicy GetTaskFailurePolicy(string workflowTypeName, string hostName).
3. TaskPolicy GetTaskTimeOutPolicy(string workflowTypeName, string hostName).
4. List<Step> GetEntryActivities(string workflowTypeName, string hostName).
5. List<Step> GetNextActivities(string workflowTypeName, string stepID, string hostName).
6. void AddWorkflow(WorkflowConfiguration workflowConfiguration, string hostName).
7. void ModifyWorkflow(WorkflowConfiguration workflowConfiguration, string hostName).
8. void DeleteWorkflow(string workflowTypeName, string hostName).
9. WorkflowConfiguration GetWorkflow(string workflowTypeName, string hostName).

Creating your own Workflow provider

To implement your own Workflow provider, we just need to create a simple .NET Core class library and add a reference to the assembly - **"PineCone.Model": "1.0.0-*".**

We need to implement the interface **PineCone.Model.IWorkflowProvider** and implement the following methods:-

1. WorkflowExecutionContext GetCurrentWorkflowContext(string hostName).
2. void Schedule(string referenceToken, Step activity, List<Step> workflows,string inputData,string hostName).
3. void CompleteWorkflow(WorkflowExecutionContext context).
4. void FailWorkflow(WorkflowExecutionContext context).
5. void MarkActivityWithRetryCount(WorkflowExecutionContext context).

6. ActivityInstance GetScheduledActivity(string hostName).
7. void CompleteActivity(ActivityInstance instance).
8. void FailActivity(ActivityInstance instance).
9. string InitiateWorkflow(string workflowTypeName, string version, string domain, string inputData).
10. string RunSynchronousWorkflow(string workflowTypeName, string version, string domain, string inputData).

Current status and future plans

At the moment PineCone.net is a solo initiative with the objective of providing a cross platform configurable workflow\integration framework that meets the business needs to develop all kinds of software for both cloud as well on premise solutions.

This is planned as a full fledged software. The sections below list what is currently available and what is planned in the future.

The future releases shall be according to demands and changes requested by the customers.

Providers

	BusinessLogic	Workflow
Current	1. AWS DynamoDB 2. MySql	1. AWS SWF 2. MySql
Future	1. SQL Server 2. Cassandra 3. DocumentDB 4. MongoDB 5. Oracle	6. SQL Server 7. Cassandra 8. DocumentDB 9. MongoDB 10. Oracle

Initiators and Activity Runners

	Initiator	Activity Runner
Current	REST Initiator (Services API)	
Future	1. AWS SQS	Synchronous:

	2. AWS SNS 3. MSMQ 4. SQL Server 5. MySql 6. Oracle 7. Cassandra 8. DocumentDB 9. MongoDB 10. Flat file 11. DynamoDB	1. REST 2. AWS SQS 3. AWS SNS 4. SQL Server 5. MySql 6. Oracle 7. Cassandra 8. DocumentDB 9. MongoDB 10. DynamoDB Asynchronous (Both incoming and outgoing): 1. AWS SQS 2. AWS SNS 3. MSMQ 4. SQL Server 5. MySql 6. Oracle 7. Cassandra 8. DocumentDB 9. MongoDB 10. Flat file 11. DynamoDB