Installation

Install orleans\_setup.msi. (v. 1.0.5)

If you already have a previous version installed, you’ll have to uninstall it first.

It relies on some environment variables, so you may have to restart visual studio or even your machine to get everything working.

Walkthrough

In this tutorial, we're going to construct multiple actors from a couple of classes with different communication interfaces. Not everything will be explained while putting together the application, we will go back and elaborate on a few things after we have it working.

Employees and Managers

This example will use the relationship of employees and managers to demonstrate the concept of multiple actors.

Open the HR solution

Have a look at the HrGrainInterfaces project and the IEmployee and IManager interfaces. They should look something like the following.

public interface IEmployee : Orleans.IGrain

{

Task<int> GetLevel();

Task Promote(int newLevel);

Task<IManager> GetManager();

Task SetManager(IManager manager);

}

and

public interface IManager : Orleans.IGrain

{

Task<IEmployee> AsEmployee();

Task<List<IEmployee>> GetDirectReports();

Task AddDirectReport(IEmployee employee);

}

In Grains, we can’t use properties. Since Properties are not meant to do I/O work, they’re not supported at all in Orleans.

It is often possible to use traditional object-oriented design methodology with Orleans, but sometimes there are reasons for not doing so. In this case, we're choosing to not rely on inheritance when defining the Manager class, even though a Manager is clearly also an Employee. The reason for this will be explained later, when we start persisting state.

Next, lets have a look at the Employee and Manager classes in HrGrains

They should look something like:

public class Employee : Orleans.Grain, Interfaces.IEmployee

{

public Task<int> GetLevel()

{

return Task.FromResult(\_level);

}

public Task Promote(int newLevel)

{

\_level = newLevel;

return TaskDone.Done;

}

public Task<IManager> GetManager()

{

return Task.FromResult(\_manager);

}

public Task SetManager(IManager manager)

{

\_manager = manager;

return TaskDone.Done;

}

private int \_level;

private IManager \_manager;

}

and

public class Manager : Orleans.Grain, IManager

{

public override Task ActivateAsync()

{

\_me = EmployeeFactory.GetGrain(this.GetPrimaryKey());

return base.ActivateAsync();

}

public Task<List<IEmployee>> GetDirectReports()

{

return Task.FromResult(\_reports);

}

public Task AddDirectReport(IEmployee employee)

{

\_reports.Add(employee);

employee.SetManager(this);

return TaskDone.Done;

}

public Task<IEmployee> AsEmployee()

{

return Task.FromResult(\_me);

}

private IEmployee \_me;

private List<IEmployee> \_reports = new List<IEmployee>();

}

A manager is expressed as an employee through composition: the manager grain has a reference to a grain representing its "employeeness."

TaskDone.Done, used in the code above, is a convenience object defined by Orleans to allow code to succinctly return an already completed Task.

In the HrSilo (Program.cs), we seed the system with a few employees, and managers, like so:

Orleans.OrleansClient.Initialize("DevTestClientConfiguration.xml");

var ids = new[] {

"42783519-d64e-44c9-9c29-399e3afaa625",

"d694a4e0-1bc3-4c3f-a1ad-ba95103622bc",

"9a72b0c6-33df-49db-ac05-14316edd332d",

"6526a751-b9ac-4881-9bfb-836ecce2ca9f",

"ae4b106f-3c96-464a-b48d-3583ed584b17",

"b715c40f-d8d2-424d-9618-76afbc0a2a0a",

"5ad92744-a0b1-487b-a9e7-e6b91e9a9826",

"e23a55af-217c-4d76-8221-c2b447bf04c8",

"2eef0ac5-540f-4421-b9a9-79d89400f7ab"

};

var e0 = EmployeeFactory.GetGrain(Guid.Parse(ids[0]));

var e1 = EmployeeFactory.GetGrain(Guid.Parse(ids[1]));

var e2 = EmployeeFactory.GetGrain(Guid.Parse(ids[2]));

var e3 = EmployeeFactory.GetGrain(Guid.Parse(ids[3]));

var e4 = EmployeeFactory.GetGrain(Guid.Parse(ids[4]));

var m0 = ManagerFactory.GetGrain(Guid.Parse(ids[5]));

var m1 = ManagerFactory.GetGrain(Guid.Parse(ids[6]));

var m0e = m0.AsEmployee().Result;

var m1e = m1.AsEmployee().Result;

m0e.Promote(10);

m1e.Promote(11);

m0.AddDirectReport(e0).Wait();

m0.AddDirectReport(e1).Wait();

m0.AddDirectReport(e2).Wait();

m1.AddDirectReport(m0e).Wait();

m1.AddDirectReport(e3).Wait();

m1.AddDirectReport(e4).Wait();

Console.WriteLine("Orleans Silo is running.\nPress Enter to terminate...");

Console.ReadLine();

In the code we have seen so far, it is noteworthy that you can send grain references (interfaces) in messages. Thus, when a direct report is added to a manager, the manager can communicate directly with the employee without calling 'GetGrain().' This ability is essential in making the programming model a smooth transition from .NET.

Now, let's add the ability for employees to send messages to each other. Expand the model like so:

public interface IEmployee : Orleans.IGrain

{

...

Task Greeting(IEmployee from, string message);

...

}

then:

public class Employee : Orleans.Grain, Interfaces.IEmployee

{

public Task Greeting(IEmployee from, string message)

{

Console.WriteLine("{0} said: {1}", from.GetPrimaryKey().ToString(), message);

return TaskDone.Done;

}

and

public class Manager : Employee, IManager

{

public async Task AddDirectReport(IEmployee employee)

{

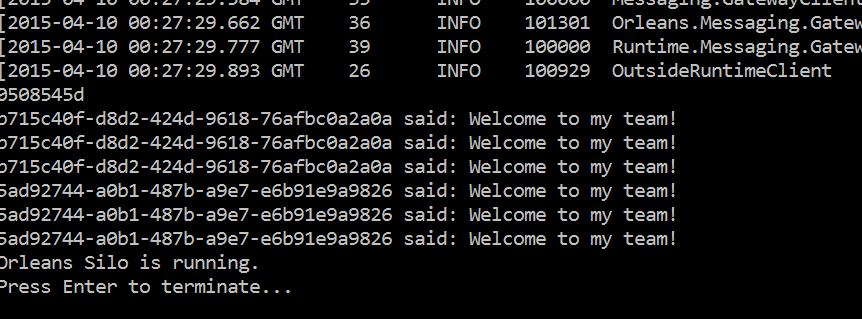
\_reports.Add(employee);

await employee.SetManager(this);

await employee.Greeting(\_me, "Welcome to my team!");

}

Executing it, you should see something like this:



The Life of an Actor

Orleans actors are virtual, meaning that even though an actor logically exists, it may not be present in memory at a given point in time. In fact, Orleans takes this concept to an extreme by claiming that all actors exist at all time (past and present) and are never created or destroyed. An actor may be inactive, such as before the first time anyone refers to it or after the last use, but is never not existing.

While counter-intuitive and, frankly, strange to most of us with a object-oriented background, this notion actually removes a source of complexity from the system. An actor is either active (present in memory) or inactive, and that status can not be observed by a client. Actor activation is on-demand, just like virtual memory is mapped on demand to physical memory.

Thus, the .NET object that holds the actor state is logically just a copy of the actor in memory while it is in the active state. Implementors of Orleans grains should refrain from writing constructors, even parameter-less ones, with any logic. That is simply the wrong way of thinking about them: grains are never created or destroyed, just moved from the inactive to the active state or vice versa.

Therefore, the implementation should catch the activation event (a call to ActivateAsync()) and perform any initialization steps necessary there. It is guaranteed to be called before any method on the grain instance is called. In the Manager grain above, it was used to establish the reference to the Employee grain.

There is also a DeactivateAsync() method, used more infrequently.

# Concurrency

Distributed applications are inherently concurrent, which leads to complexity. One of the things that makes the actor model special and productive is that it helps reduce some of the complexities of having to grapple with concurrency.

Actors accomplish this in two ways: \* By providing single-threaded access to the internal state of an actor instance. \* By not sharing data between actor instances except via message-passing.

In this tutorial, we will examine both of these aspects of the programming model.

## Turn-based Execution

The idea behind the single-threaded execution model for actors is that the invokers (remote) take turns "calling" its methods. Thus, a message coming to actor B from actor A will placed in a queue and the associated handler is invoked only when all prior messages have been serviced.

This allows us to avoid all use of locks to protect actor state, as it is inherently protected against data races. However, it may also lead to problems when messages pass back and forth and the message graph forms cycles. If A sends a message to B from one of its methods and awaits its completion, and B sends a message to A, also awaiting its completion, the application will quickly lock up.

To illustrate, let's go back to the code that was established in the tutorial on collections of actors and modify it to demonstrate how things can go bad by creating a trivial cycle in the messaging graph: when an employee receives a greeting, he sends another greeting back to the sender and waits for the acknowledgment. This will send a back-and-forth series of messages, until we get to 3.

First create a class in the interface project which we'll use to send the greetings around:

public class GreetingData

{

public long From { get; set; }

public string Message { get; set; }

public int Count { get; set; }

}

From will be the sender of the message (the ID of the grain), Message will be the message text, and Count will be the number of times the message has been sent back and forth. This stops us from getting a stack overflow.

We need to modify the arguments of Greeting on the Employee interface to :

Task Greeting(GreetingData data);

We need to update the implementation accordingly:

public async Task Greeting(GreetingData data)

{

Console.WriteLine("{0} said: {1}", data.From, data.Message);

// stop this from repeating endlessly

if (data.Count >= 3) return;

// send a message back to the sender

var fromGrain = EmployeeFactory.GetGrain(data.From);

await fromGrain.Greeting(new GreetingData {

From = this.GetPrimaryKeyLong(),

Message = "Thanks!",

Count = data.Count + 1 });

}

We'll also update the Manager class, so it send the new message object:

public async Task AddDirectReport(IEmployee employee)

{

\_reports.Add(employee);

await employee.SetManager(this);

await employee.Greeting(new GreetingData {

From = this.GetPrimaryKeyLong(),

Message = "Welcome to my team!" });

}

Now the Employee sends a message back to the manager, saying "Thanks!".

Let's add some simple client code to add a direct report to a manager by commenting out the previous seeding code and adding:

var e0 = EmployeeFactory.GetGrain(0);

var m1 = ManagerFactory.GetGrain(1);

m1.AddDirectReport(e0);

When we run this code, the first "Thanks!" greeting is received. However, when this message is responded to this we get a 30 second pause, then warnings appear in the log and we're told the grain is about to break it's promise.

1 said: Welcome to my team!

0 said: Thanks!

[2014-03-12 15:25:37.398 GMT 31 WARNING 100157 CallbackData 127.0.0.1:11111] Response did

not arrive on time in 00:00:30 for message: Request

S127.0.0.1:11111:132333898\*grn/906ECA4C/00000001@68e2b3ab->S127.0.0.1:11111:132333898\*grn/D9BB797F/00000000@c24c4187 #13: MyGrainInterfaces1.IEmployee:Greeting(). Target History is: <S127.0.0.1:11111:132333898:\*grn/D9BB797F/00000000:@c24c4187>.

About to break its promise.

[2014-03-12 15:25:37.398 GMT 27 WARNING 100157 CallbackData 127.0.0.1:11111] Response did not arrive on time in 00:00:30 for message: Request S127.0.0.1:11111:132333898\*grn/D9BB797F/00000000@c24c4187->S127.0.0.1:11111:132333898\*grn/D9BB797F/00000001@afc70cb4 #14: MyGrainInterfaces1.IEmployee:Greeting(). Target History is: <S127.0.0.1:11111:132333898:\*grn/D9BB797F/00000001:@afc70cb4>. About to break its promise.

[2014-03-12 15:25:37.407 GMT 28 WARNING 100157 CallbackData 127.0.0.1:11111] Response did not arrive on time in 00:00:30 for message: Request S127.0.0.1:11111:132333898\*grn/D9BB797F/00000001@afc70cb4->S127.0.0.1:11111:132333898\*grn/D9BB797F/00000000@c24c4187 #15: MyGrainInterfaces1.IEmployee:Greeting(). Target History is: <S127.0.0.1:11111:132333898:\*grn/D9BB797F/00000000:@c24c4187>. About to break its promise.

An exception is then thrown in the client code.

We've created a deadlock. Grain 0 sends a message to grain 1. In that call grain 1 sends a message back to grain 0. However, grain 0 can't process it because it's awaiting the first message, so it gets queued. The await can't complete until the second message is returned, so we've entered a state that we can't escape from. Orleans waits for 30 seconds, then kills the request.

Orleans offers us a way to deal with this, by marking the grain [Reentrant], which means that additional calls may be made while the grain is waiting for a task to complete, resulting in interleaved execution.

[Reentrant]

public class Employee : Orleans.Grain, Interfaces.IEmployee

{

public async Task Greeting(GreetingData data)

{

Console.WriteLine("{0} said: {1}",

data.From.GetPrimaryKey().ToString(),

data.Message);

await data.From.Greeting(

new GreetingData {

From = EmployeeFactory.Cast(this.AsReference()),

Message = "Hi yourself!" });

}

We see that the sample works, and Orleans is able to interleave the grain calls:

1 said: Welcome to my team!

0 said: Thanks!

1 said: Thanks!

0 said: Thanks!

Messages Messages are simply data passed from one actor to, we just created the GreetingData class to do just this.

In .NET, most objects are created from a class of some sort and are passed around by reference, something that doesn't work well with concurrency, and definitely not with distribution.

When Orleans sends a message from one grain to another, it creates a deep copy of the object, and provides the copy to the second grain, and not the object stored in the first grain. This prohibits the mutation of state from one grain to another, one of the main tenants in the actor model is that state shouldn't be shared, and message passing is the only mechanism for exchanging data.

When the grains are in different silos, the object model is serialized to a binary format, and sent over the wire.

However, this deep copy process is expensive, and if you promise not to modify the message, then for communication with grains within a silo, it's unnecessary.

If you indicate to Orleans that you are not going to modify the object (i.e. it's immutable) then it can skip the deep copy step, and it will pass the object by reference. There's no way Orleans or C# can stop you from modifying the state, you have to be disciplined.

Immutability is indicated with a the [Immutable] attribute on the class:

[Immutable]

public class GreetingData

{

public long From { get; set; }

public string Message { get; set; }

public int Count { get; set; }

}

No other code change is required, this is just a signal to give to Orleans to tell it you’re not going to modify this object.

# Declarative Persistence

It is sometimes that case that some of the state you are accumulating belongs in some form of permanent storage, so that it can survive a silo shutdown, or a grain migrating from one silo to another for load-balancing or a complete restart/shutdown of the service. What we have seen so far will not support such situations.

Fortunately, Orleans offers a simple declarative model for identifying the state that needs to be stored in a permanent location, while leaving the decision when to save and restore state under programmatic control. You are not required to use the declarative persistence mechanism and can still access storage directly from your grain code, but it’s a nice way to save you some boilerplate code and build applications that are portable across various storage services.

## Getting Started

We'll continue to build on our employee-and-manager sample.

Uncomment the previously commented seed sections in the HrSilo Program.cs file:

static void Main(string[] args)

{

...

Orleans.OrleansClient.Initialize("DevTestClientConfiguration.xml");

var ids = new string[] {

"42783519-d64e-44c9-9c29-399e3afaa625",

"d694a4e0-1bc3-4c3f-a1ad-ba95103622bc",

"9a72b0c6-33df-49db-ac05-14316edd332d",

"6526a751-b9ac-4881-9bfb-836ecce2ca9f",

"ae4b106f-3c96-464a-b48d-3583ed584b17",

"b715c40f-d8d2-424d-9618-76afbc0a2a0a",

"5ad92744-a0b1-487b-a9e7-e6b91e9a9826",

"e23a55af-217c-4d76-8221-c2b447bf04c8",

"2eef0ac5-540f-4421-b9a9-79d89400f7ab"

};

var e0 = EmployeeFactory.GetGrain(Guid.Parse(ids[0]));

var e1 = EmployeeFactory.GetGrain(Guid.Parse(ids[1]));

var e2 = EmployeeFactory.GetGrain(Guid.Parse(ids[2]));

var e3 = EmployeeFactory.GetGrain(Guid.Parse(ids[3]));

var e4 = EmployeeFactory.GetGrain(Guid.Parse(ids[4]));

var m0 = ManagerFactory.GetGrain(Guid.Parse(ids[5]));

var m1 = ManagerFactory.GetGrain(Guid.Parse(ids[6]));

...

}

Next, we'll do some silo configuration, in order to configure the storage provider that will give us access to persistent storage. The silo host project includes a configuration file 'DevTestServerConfiguration.xml' which is where we find the following section:

<OrleansConfiguration xmlns="urn:orleans">

<Globals>

<StorageProviders>

<Provider Type="Orleans.Storage.MemoryStorage" Name="MemoryStore" />

<!-- To use Azure storage, uncomment one of the following elements: -->

<!--

<Provider Type="Orleans.Storage.AzureTableStorage"

Name="AzureStore"

DataConnectionString="UseDevelopmentStorage=true" />

-->

<!--

<Provider Type="Orleans.Storage.AzureTableStorage"

Name="AzureStore"

DataConnectionString="[removed for brevity]" />

-->

</StorageProviders>

The 'MemoryStorage' provider is fairly uninteresting, since it doesn't actually provide any permanent storage; it's intended for debugging persistent grains while having no access to a persistent store. If you have the latest SDK of Azure and the Azure Storage emulator installed, you could start it and use the uncomment the provider. If you don’t have it installed, you could use the MemoryStorage provider.

With one of those enabled, we're ready to tackle the grain code.

Note: The built-in storage provider classes Orleans.Storage.MemoryStorage and Orleans.Storage.AzureTableStorage are in the OrleansProviders.dll assembly, so make sure that DLL is referenced in your silo worker role project with CopyLocal=True. You can find it on nuget.

## Declaring State

Identifying that a grain should use persistent state takes three steps: declaring an interface for the state, changing the grain base class, and identifying the storage provider.

The first step, declaring an interface, simply means identifying the information of an actor that should be persisted and creating what looks like a record of the persistent data -- each state component is represented by a property with a getter and a setter.

For employees, we want to persist all the state:

public interface IEmployeeState : IGrainState

{

int Level { get; set; }

IManager Manager { get; set; }

}

and for managers, we must store the direct reports, but the "\_me" reference may continue to be created during activation.

public interface IManagerState : IGrainState

{

List<IEmployee> Reports { get; set; }

}

Then, we change the grain class declaration to identify the state interface and remove the variables that we want persisted. Make sure to remove level, and manager from the Employee class and \_reports from the Manager class.

We also add an attribute to identify the storage provider:

[StorageProvider(ProviderName = "MemoryStore")]

public class Employee : Orleans.Grain<IEmployeeState>, Interfaces.IEmployee

and

[StorageProvider(ProviderName="MemoryStore")]

public class Manager : Orleans.Grain<IManagerState>, IManager

At risk of stating the obvious, the name of the storage provider attribute should match the name in the configuration file. This indirection is what allows you to delay choices around where to store grain state until deployment.

Given these declarative changes, the grain should no longer rely on a private fields to keep compensation level and manager. Instead, the grain base class gives us access to the state via a 'State' property that is available to the grain.

For example:

public Task SetManager(IManager manager)

{

State.Manager = manager;

return TaskDone.Done;

}

## Controlling Checkpoints

The question that remains is when the persistent state gets saved to the storage provider.

One choice that the Orleans designers could have made would be to have the runtime save state after every method invocation, but that turns out to be undesirable because it is far too conservative -- not all invocations will actually modify the state on all invocations, and some will never modify it. Rather than employing a complex system to evaluate state differentials after each method, Orleans asks the grain developer to add the necessary logic to determine whether state needs to be saved or not.

Saving the state using the storage provider is easily accomplished by calling 'State.WriteStateAsync()'.

Thus, the final version of the 'Promote()' and 'SetManager()' methods looks like this:

public Task Promote(int newLevel)

{

State.Level = newLevel;

return State.WriteStateAsync();

}

public Task SetManager(IManager manager)

{

State.Manager = manager;

return State.WriteStateAsync();

}

In the Manager class, there's only one method that need to be modified to write out data, 'AddDirectReport().' It should look like this:

public async Task AddDirectReport(IEmployee employee)

{

State.Reports.Add(employee);

await employee.SetManager(this);

var data = await employee.Greeting(

new GreetingData { From = \_me, Message = "Welcome to my team!" });

Console.WriteLine("{0} said: {1}",

data.From.ToString(),

data.Message);

await State.WriteStateAsync();

}

Let's try this out!

## Mixing Things

A grain may contain a combination of persisted and transient state. Any transient state should be represented by private fields in the grain class. A common use for mixing the two is to cache some computed version of the persisted state in private fields while it is present in memory. For example, a stack of elements may be externally represented as a List, but internally, as a Stack.

In the case of our Manager class, the \_me field is simply a cached value, something we don't even need to keep as a field in the first place, it can be created any time we need it, but since it's going to be a commonly used value, it's worth keeping it around in a transient field.