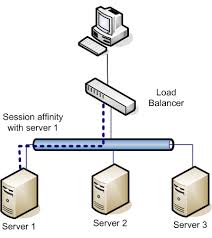
**Distributed applications** (**distributed** apps) are **applications** or software that runs on multiple computers within a network at the same time and can be stored on servers or with cloud computing.



<https://dotnet.github.io/orleans/>

silo : a [large](https://dictionary.cambridge.org/dictionary/english/large), round [tower](https://dictionary.cambridge.org/dictionary/english/tower) on a [farm](https://dictionary.cambridge.org/dictionary/english/farm) for [storing](https://dictionary.cambridge.org/dictionary/english/store) [grain](https://dictionary.cambridge.org/dictionary/english/grain) or [winter](https://dictionary.cambridge.org/dictionary/english/winter) [food](https://dictionary.cambridge.org/dictionary/english/food) for [cattle](https://dictionary.cambridge.org/dictionary/english/cattle):

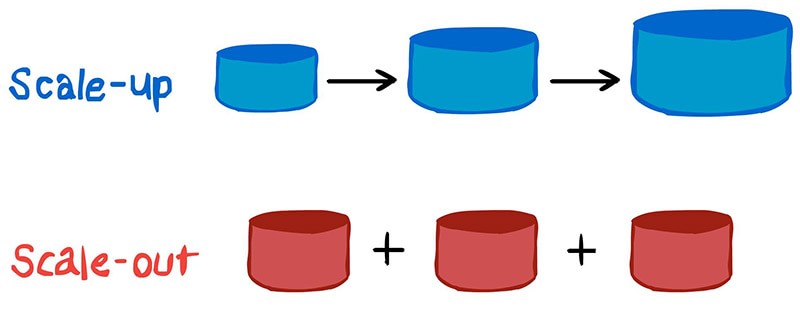
grain : seeds

scaffold : a [frame](https://dictionary.cambridge.org/dictionary/english/frame) that [hangs](https://dictionary.cambridge.org/dictionary/english/hang) on the [side](https://dictionary.cambridge.org/dictionary/english/side) of a [building](https://dictionary.cambridge.org/dictionary/english/building), [ship](https://dictionary.cambridge.org/dictionary/english/ship), etc. for [people](https://dictionary.cambridge.org/dictionary/english/people) to [work](https://dictionary.cambridge.org/dictionary/english/work) from

scalability : the [ability](https://dictionary.cambridge.org/dictionary/english/ability) of a [business](https://dictionary.cambridge.org/dictionary/english/business) or [system](https://dictionary.cambridge.org/dictionary/english/system) to [grow](https://dictionary.cambridge.org/dictionary/english/grow) [larger](https://dictionary.cambridge.org/dictionary/english/large):

intuitive : an [ability](https://dictionary.cambridge.org/dictionary/english/ability) to [understand](https://dictionary.cambridge.org/dictionary/english/understand) or [know](https://dictionary.cambridge.org/dictionary/english/know) something [immediately](https://dictionary.cambridge.org/dictionary/english/immediately) [based](https://dictionary.cambridge.org/dictionary/english/based) on [your](https://dictionary.cambridge.org/dictionary/english/your) [feelings](https://dictionary.cambridge.org/dictionary/english/feeling) [rather](https://dictionary.cambridge.org/dictionary/english/rather) than [facts](https://dictionary.cambridge.org/dictionary/english/fact):

**Scale out** is a type of capacity expansion concentrating on the addition of new hardware resources instead of increasing the capacity of already available hardware resources such as storage or processing silos.



<https://codeopinion.com/practical-microsoft-orleans/>

<https://bitbucket.org/dandago/gigilabs/src/master/>

<http://gigi.nullneuron.net/gigilabs/getting-started-with-microsoft-orleans-2-0-in-net-core/>

**Orleans** is a framework that provides a straight-forward approach to building distributed high-scale computing applications, without the need to learn and apply complex concurrency or other scaling patterns.

Implementation of Orleans is based on the Actor Model that’s been around since 1970s. However, unlike actors in more traditional actor systems such as Erlang or Akka, [Orleans Grains](http://dotnet.github.io/orleans/Documentation/Getting-Started-With-Orleans/Grains.html) are virtual actors. The biggest difference is that physical instantiations of grains are completely abstracted away and are automatically managed by the Orleans runtime. The Virtual Actor Model is much more suitable for high-scale dynamic workloads like cloud services and is the major innovation of Orleans

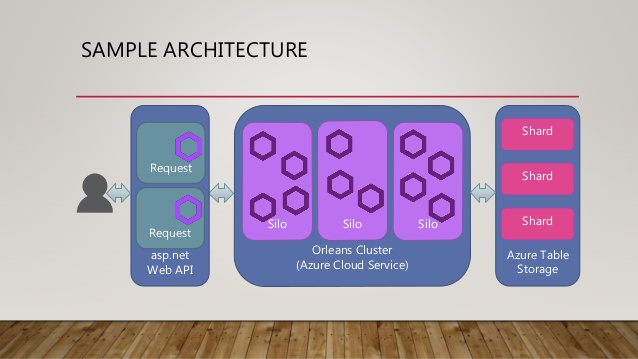
#### Grains are the key primitives of the Orleans programming model. Grains are the building blocks of an Orleans application, they are atomic units of isolation, distribution, and persistence. Grains are objects that represent application entities. Just like in the classic Object Oriented Programming, a grain encapsulates state of an entity and encodes its behavior in the code logic. Grains can hold references to each other and interact by invoking each other’s methods exposed via interfaces.

One other interesting note about grains is that they are automatically managed by Orleans.  You don’t have to worry about instantiating or managing them.   Orleans terms them virtual actors as apposed to traditional actors.

#### Silos Where do Grains live? Silos of course.

Silos are what host and execute Grains.  Again, Grains are your objects that expose behavior and encapsulate state.  Orleans creates your grains in the Silo and executes them here.  Your client code will reference only interfaces that your grains implement.  Don’t worry, sample code coming up.

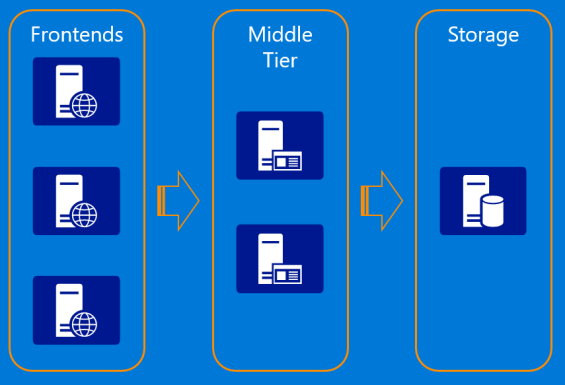
Oh and of course, you can also have a cluster of silos.



**Background**

Cloud applications and services are inherently parallel and distributed. They are also interactive and dynamic; often requiring near real time direct interactions among cloud entities. Such applications are very difficult to build today. The development process demands expert level programmers and typically requires expensive iterations of the design and the architecture, as the workload grows.

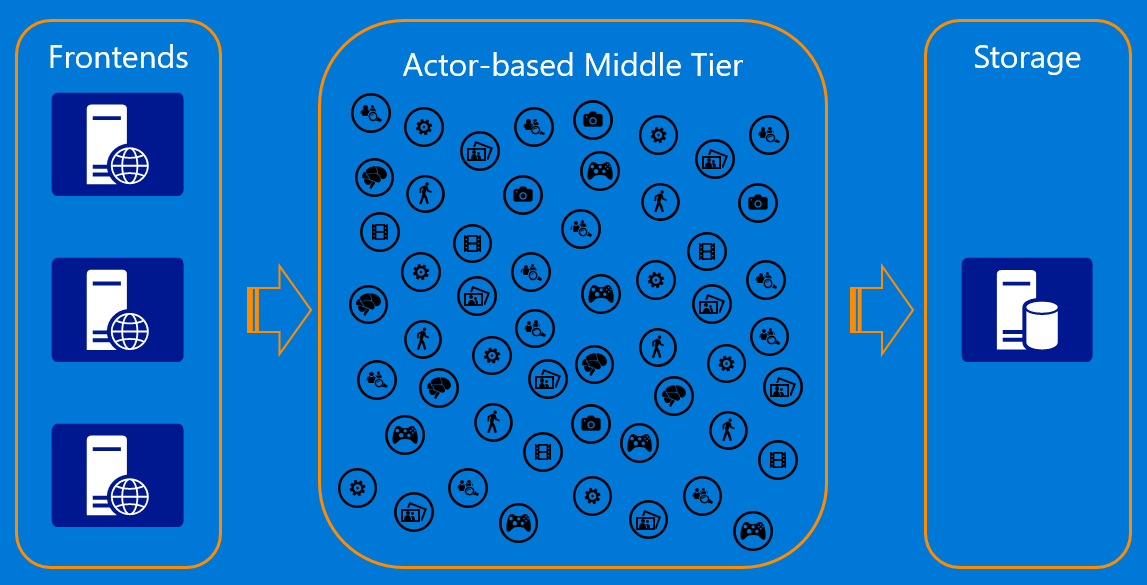
Most of today’s high scale properties are built as a composition of stateless n-tier services with most of the application logic residing in the middle tier.



While the model allows scaling out by adding more servers to the middle tier, it is constrained by the performance and scalability of the storage layer because most requests coming to the middle tier from the frontend web servers require one or more reads from storage. Updates are even more complicated and prone to concurrency issues and conflicts due to lack of coordination among the middle tier servers. It often requires caching in the stateless layer to get acceptable performance, adding complexity and introducing cache consistency issues. The other problem with the stateless n-tier model is that it doesn't support horizontal communications well among individual application entities exposed by the middle tier, which makes it hard to implement complex business logic with multiple entities performing individual operations as part of processing a request.

## Orleans as a Stateful Middle Tier

Orleans provides an intuitive way of building a stateful middle tier, where various business logic entities appear as sea of isolated globally addressable .NET objects (grains) of different application defined types distributed across a cluster of servers (silos).



A grain type is a simple .NET class that implements one or more application-defined grain interfaces. Individual grains are instances of application-defined grain classes that get automatically created by the Orleans runtime on servers on an as-needed basis to handle requests for those grains. Grains naturally map to most application entities, such as users, devices, sessions, inventories, and orders. This makes it very easy to build business logic that is object-oriented but scales transparently across a cluster of servers. Each grain has a stable logical identity (key) within its grain type chosen by the application logic, for example, user email or device ID or inventory SKU code. Orleans guarantees single-threaded execution of each individual grain, hence protecting the application logic from perils of concurrency and races. In the world of microservices, Orleans is used as a framework for implementing a microservice that can be deployed and managed by a microservices deployment/management solution of developer's choice.

# GETTING STARTED WITH MICROSOFT ORLEANS 2.0 IN .NET CORE

[APRIL 10, 2018](http://gigi.nullneuron.net/gigilabs/getting-started-with-microsoft-orleans-2-0-in-net-core/) [GIGI](http://gigi.nullneuron.net/gigilabs/author/gigi/) [4 COMMENTS](http://gigi.nullneuron.net/gigilabs/getting-started-with-microsoft-orleans-2-0-in-net-core/#comments)

[Microsoft Orleans](https://dotnet.github.io/orleans/index.html) 2.0 was [released](https://twitter.com/msftorleans/status/979566567435395072) less than two weeks ago. The biggest win here is .NET Core/Standard support, meaning that **Orleans is cross-platform**. In this article, we’ll see how to quickly get up and running with Orleans 2.0.

The configuration and hosting APIs have changed considerably, so the instructions here won’t work for earlier versions. See my old “[*Getting Started with Microsoft Orleans*](http://gigi.nullneuron.net/gigilabs/getting-started-with-microsoft-orleans/)” article from November 2016 if you’re running Orleans 1.4. Orleans 1.5 is also different so you’ll need to check the documentation for that.

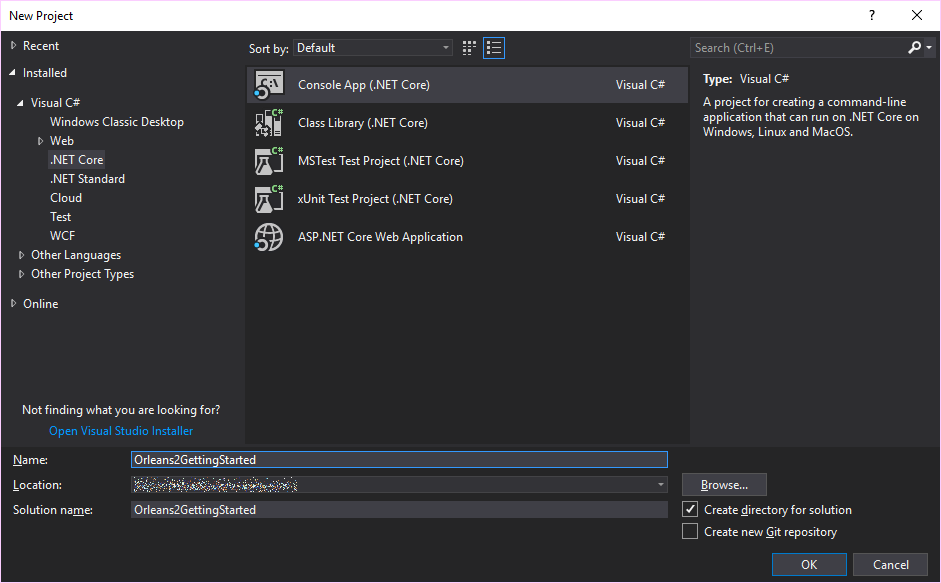
In order to keep this article practical and concise, it is necessary to limit its scope. We will not be covering what Orleans is or what it is used for. We will also not create a full project structure that is typical in Orleans solutions. Instead, we’ll keep it simple so that in a short time we have a starting point to explore what Orleans has to offer.

Tip: Use Ctrl+. (Control dot) to resolve namespaces in Visual Studio.

The source code for this article is the ***Orleans2GettingStarted*** folder in the *[Gigi Labs BitBucket repository](https://bitbucket.org/dandago/gigilabs/src)*.

**Update 30th June 2018**: The source code for this article needs a little adjusting, in order to [gracefully stop the silo](https://dotnet.github.io/orleans/Documentation/Deployment-and-Operations/SiloShutdown.html) and [gracefully close the client](https://github.com/dotnet/orleans/issues/4436). Counterintuitively, directly disposing a silo or client is non-graceful and is generally discouraged.

# .NET Core Project

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans2-start-core-proj.png)

To demonstrate the fact that Orleans 2.0 really supports .NET Core, we’ll create a .NET Core console app and set up everything in it. To keep things simple, we’ll run both the client and the silo (server) from this same application.

# Install Packages

There are a few [packages](https://dotnet.github.io/orleans/Documentation/Getting-Started-With-Orleans/NuGets.html) we’ll need:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Install-Package Microsoft.Orleans.Server  Install-Package Microsoft.Orleans.Client  Install-Package Microsoft.Orleans.OrleansCodeGenerator.Build  Install-Package Microsoft.Extensions.Logging.Console  Install-Package OrleansDashboard |

Here is a summary of what each of these does:

1. Used by Orleans silo (server).
2. Used by Orleans client.
3. Required for build-time [code generation](https://dotnet.github.io/orleans/Documentation/Advanced-Concepts/Codegen.html). [Bad things](https://github.com/dotnet/orleans/issues/4417) happen if you don’t include it.
4. Optional, but allows us to set up logging to console to see what Orleans is doing.
5. Optional, but allows us to visualise the operation of silos and grains.

**Update 28th April 2018**: if you’re running on Windows, you can also get [CPU and memory telemetry](https://github.com/OrleansContrib/OrleansDashboard/issues/122) in the Orleans dashboard by installing the **Microsoft.Orleans.OrleansTelemetryConsumers.Counters** package.

# Grain and Grain Interface

We’ll add a simple grain class to the project in order to have a minimal example. Since grains are independent of each other, Internet of Things (IoT) scenarios fit very nicely. Imagine we have a number of temperature sensors deployed in different places. Each one has an ID, and periodically submits temperature readings to a corresponding grain in the Orleans cluster:

|  |  |
| --- | --- |
|  | public class TemperatureSensorGrain : Grain, ITemperatureSensorGrain  {      public Task SubmitTemperatureAsync(float temperature)      {          long grainId = this.GetPrimaryKeyLong();          Console.WriteLine($"{grainId} received temperature: {temperature}");            return Task.CompletedTask;      }  } |

We’re not doing anything special here. We write out the grain ID and the value we received just so we see something going on in the console. It is important that we inherit from the Grain base class, and that all our methods return Task.

We also need a grain interface:

|  |  |
| --- | --- |
|  | public interface ITemperatureSensorGrain : IGrainWithIntegerKey  {      Task SubmitTemperatureAsync(float temperature);  } |

The interface must inherit from an Orleans-defined interface that tells what type of grain ID (key) it will use. Our grains will have an ID of type long (that’s some misleading naming in the interface), but there are other options including GUID or string.

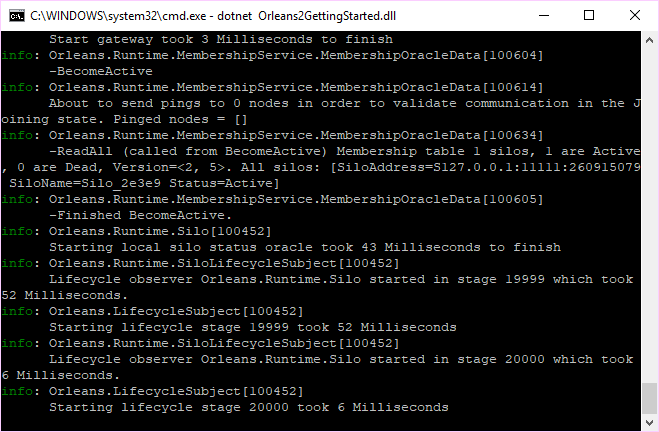
# Silo

Taking a look at the [Hello World sample](https://dotnet.github.io/orleans/Documentation/Samples-Overview/Hello-World.html) gives an idea of how to set up minimal silo and client. Since this code is going to be async, we’ll need use C# 7.1+ (to support async/await in Main()) or use a workaround. See the last section of “[Working with Asynchronous Methods in C#](http://gigi.nullneuron.net/gigilabs/working-with-asynchronous-methods-in-c/)” for how this is done (quick tip: Project Properties -> Build -> Advanced… -> C# latest minor version (latest)).

We can adapt the code from the Hello World sample to run a simple silo:

|  |  |
| --- | --- |
|  | static async Task Main(string[] args)  {      var siloBuilder = new SiloHostBuilder()          .UseLocalhostClustering()          .Configure<ClusterOptions>(options =>          {              options.ClusterId = "dev";              options.ServiceId = "Orleans2GettingStarted";          })          .Configure<EndpointOptions>(options =>              options.AdvertisedIPAddress = IPAddress.Loopback)          .ConfigureLogging(logging => logging.AddConsole());        using (var host = siloBuilder.Build())      {          await host.StartAsync();            Console.ReadLine();      }  } |

Here, we are setting up a local cluster for development. Thanks to the console logging package we installed earlier and the ConfigureLogging() call above, we can see what Orleans is up to:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans2-start-first-silo.png)

What is being written out is not important at this stage. The important thing is that the Orleans silo is running.

# Client

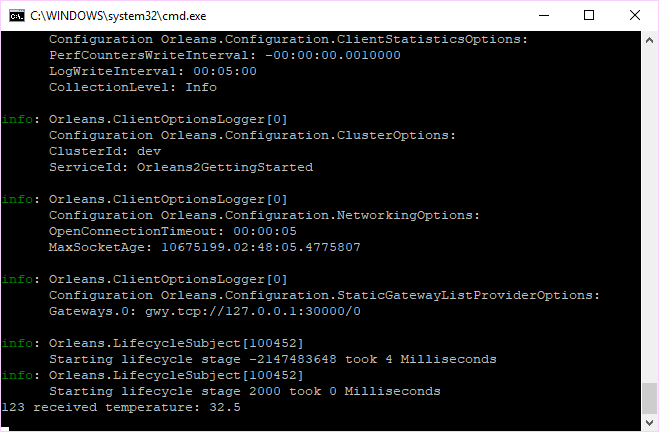
The same [Hello World sample](https://dotnet.github.io/orleans/Documentation/Samples-Overview/Hello-World.html) also shows us how to set up a client that connects to the silo. This usually serves as a gateway between the outside world and the Orleans cluster. It could be a Web API, Windows service, etc; but here it will just be in the same console app as the silo.

We’ll wait to start the client after the silo has started. This is easy to do in our case because both are in the same application.

|  |  |
| --- | --- |
|  | using (var host = siloBuilder.Build())  {      await host.StartAsync();        var clientBuilder = new ClientBuilder()          .UseLocalhostClustering()          .Configure<ClusterOptions>(options =>          {              options.ClusterId = "dev";              options.ServiceId = "Orleans2GettingStarted";          })          .ConfigureLogging(logging => logging.AddConsole());        using (var client = clientBuilder.Build())      {          await client.Connect();            var sensor = client.GetGrain<ITemperatureSensorGrain>(123);          await sensor.SubmitTemperatureAsync(32.5f);            Console.ReadLine();      }  } |

The setup for the client is very similar to that of the silo, and quite straightforward since we are using the default localhost configurations. One thing you’ll notice is the unfortunate [inconsistent naming](https://github.com/dotnet/orleans/issues/4436) between host.StartAsync() and client.Connect(); the latter lacks the –Async() suffix, even though it also returns a Task.

If we run this code, we see that the code in the grain is getting executed, and we see the temperature reading in the console at the end:

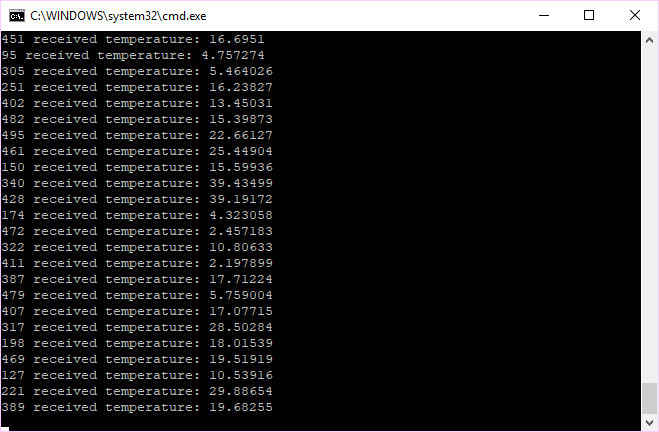
[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans2-start-client-output.png)

# Dashboard

Although it works, this example is really boring. We essentially have a Hello World here, styled for IoT. Let’s change the client code to generate some load instead:

|  |  |
| --- | --- |
|  | using (var client = clientBuilder.Build())  {      await client.Connect();        var random = new Random();      string sky = "blue";        while (sky == "blue") // if run in Ireland, it exits loop immediately      {          int grainId = random.Next(0, 500);          double temperature = random.NextDouble() \* 40;          var sensor = client.GetGrain<ITemperatureSensorGrain>(grainId);          await sensor.SubmitTemperatureAsync((float)temperature);      }  } |

Now we can see the silo brimming with activity, but only in the console:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans2-start-client-load.png)

Now, wouldn’t it be nice if we could see some graphs showing what our grains and silos are doing? As it turns out, we can do that by setting up the [Orleans Dashboard](https://github.com/OrleansContrib/OrleansDashboard), a community-contributed admin dashboard for Microsoft Orleans.

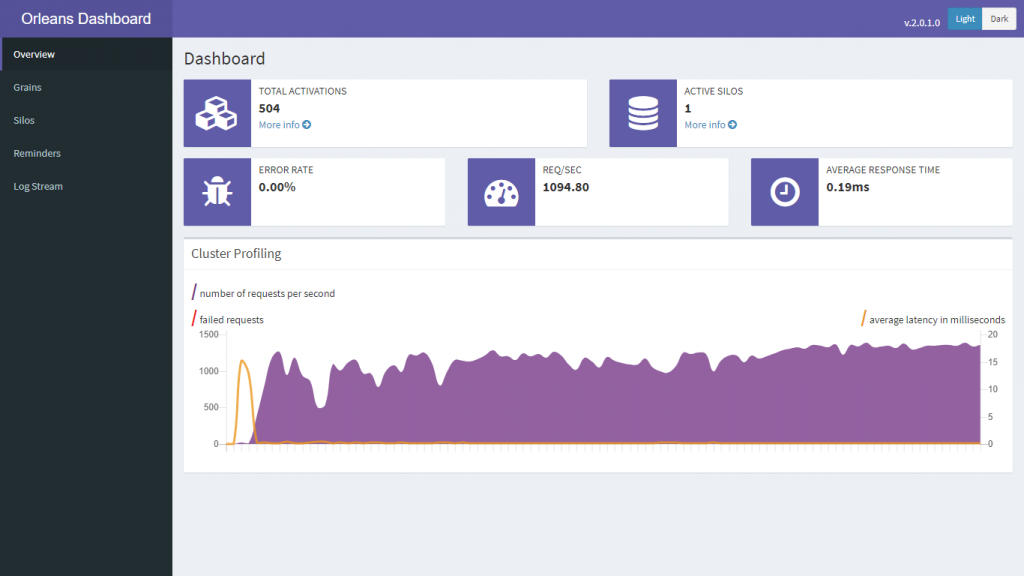
We’ve already installed the package for it, so all we need to do is add it to the silo configuration:

|  |  |
| --- | --- |
|  | var siloBuilder = new SiloHostBuilder()      .UseLocalhostClustering()      .UseDashboard(options => { })      .Configure<ClusterOptions>(options =>      {          options.ClusterId = "dev";          options.ServiceId = "Orleans2GettingStarted";      })      .Configure<EndpointOptions>(options =>          options.AdvertisedIPAddress = IPAddress.Loopback)      .ConfigureLogging(logging => logging.AddConsole()); |

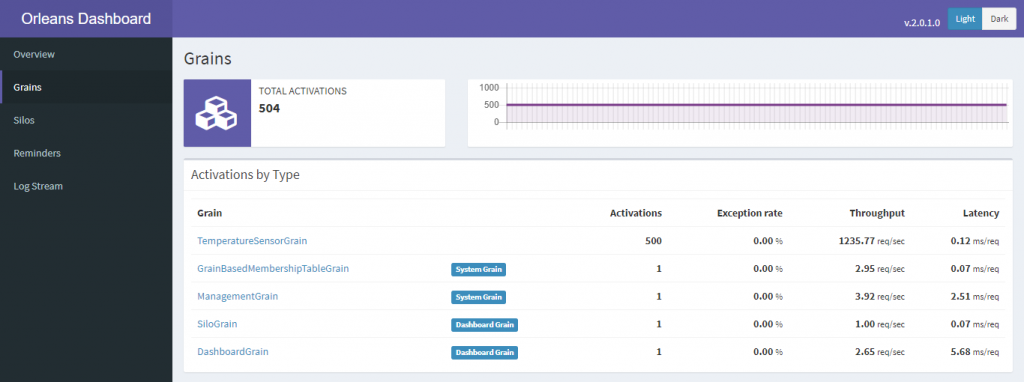
That sets up the dashboard with all default values (port 8080, no username/password) which you can always change if you need to.

So now, if we run the application again and open localhost:8080 in a browser window, we can get some pretty visualisations.

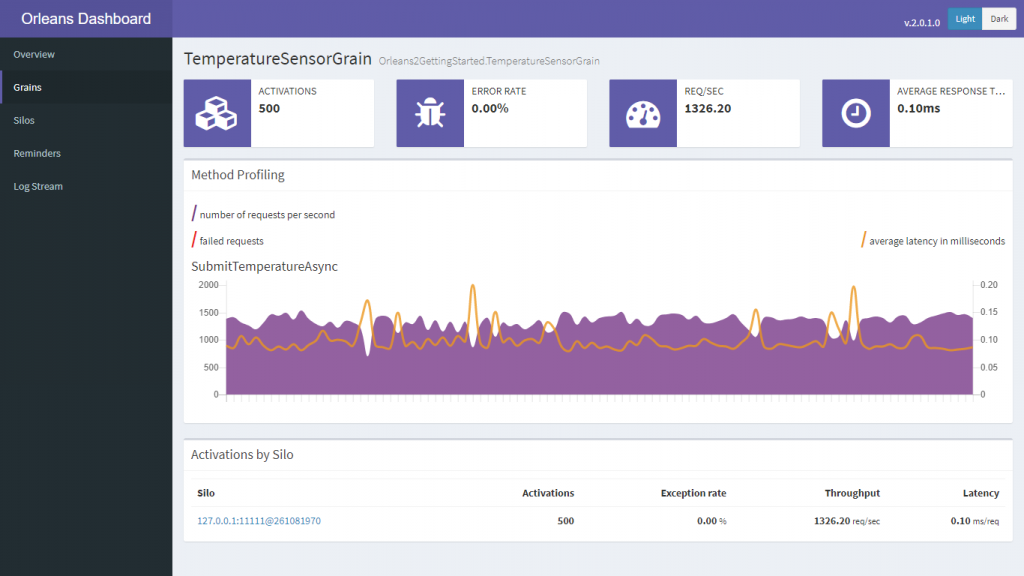
Here’s the high-level view of the cluster:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-dashboard-1.png)

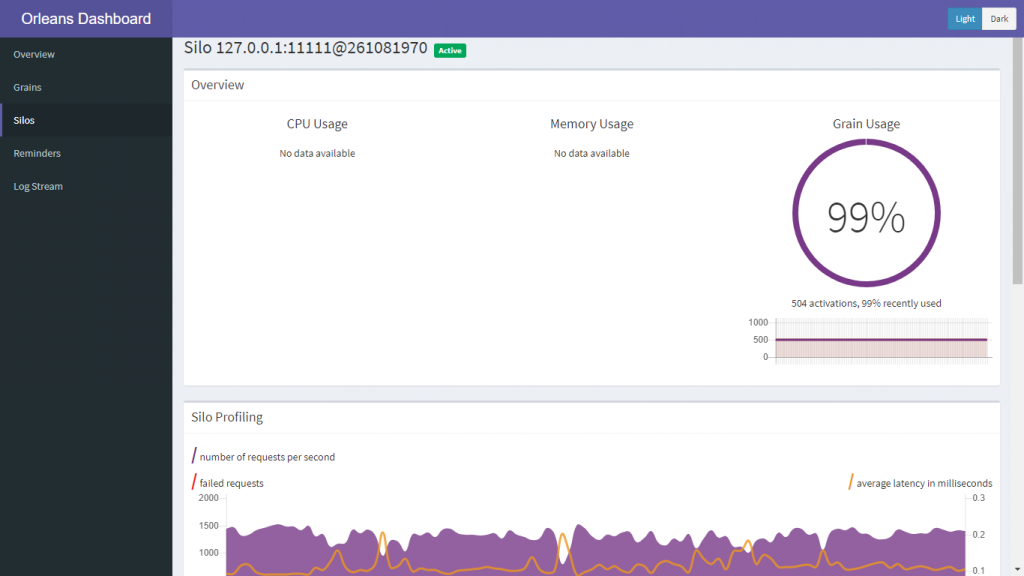
And here’s a view of the grains that are running. You’ll see we have 500 instances of our TemperatureSensorGrain, which corresponds to the range of grainIds we’re generating at random as we generate load. You’ll also see some internal system-related grains:

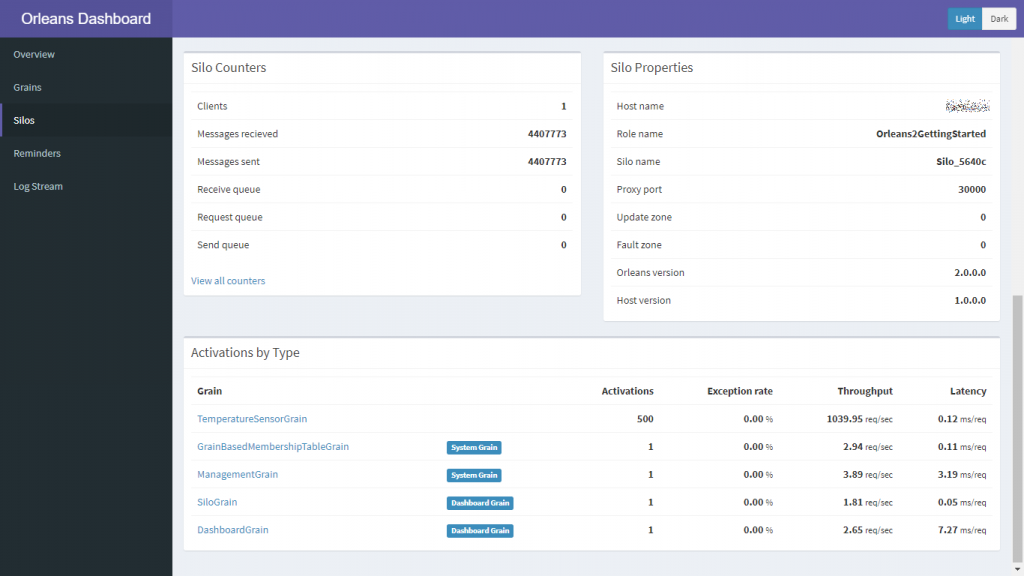
[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-dashboard-2.png)

Here’s a view of the grain itself, and the methods being called on it:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-dashboard-3.png)

We can also get a view of the silo:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-dashboard-4.png)

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-dashboard-5.png)

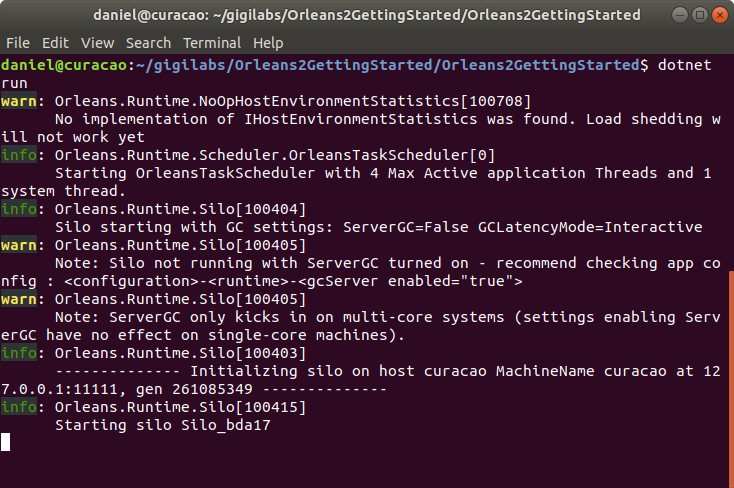
We haven’t covered everything the dashboard gives you, but you can already see that it gives a lot of visibility into what’s going on. It’s great to track errors, slow requests, throughput, etc.

# Linux

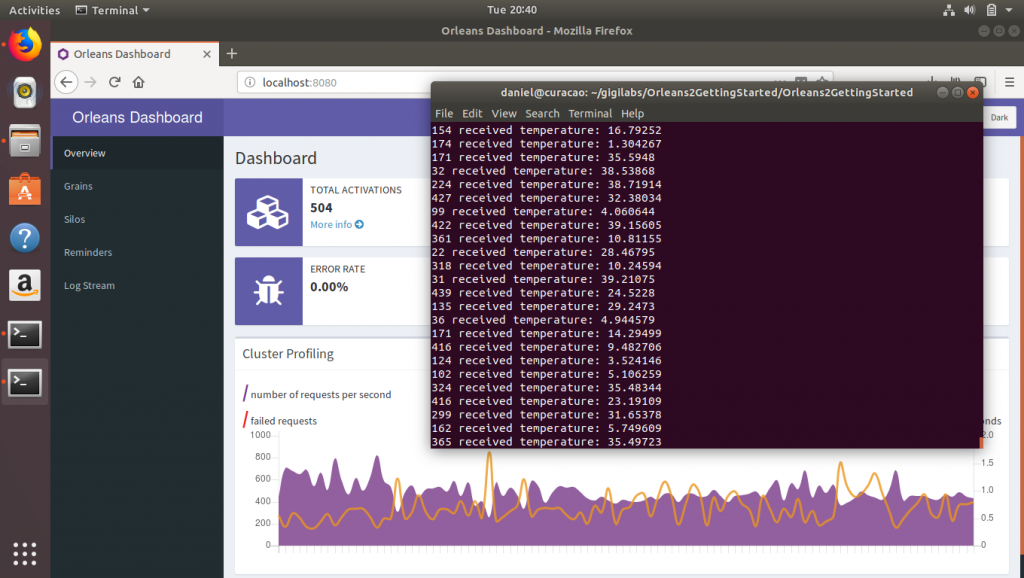
So now we have Orleans in a .NET Core project on Windows. That’s great, but the real benefit of Orleans supporting .NET Core is cross-platform deployment. So after [installing .NET Core on a Linux machine](https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites?tabs=netcore2x) (I’m using Ubuntu 17.10.1 here), let’s grab the code and run Orleans:

|  |  |
| --- | --- |
|  | git clone https://bitbucket.org/dandago/gigilabs.git  cd gigilabs  cd Orleans2GettingStarted  cd Orleans2GettingStarted  dotnet run |

Orleans has no problem starting up on this Ubuntu VM:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-ubuntu-1.png)

And here we can see Orleans running with the Orleans Dashboard in the background:

[](http://gigi.nullneuron.net/gigilabs/wp-content/uploads/2018/04/orleans-start-ubuntu-2.png)

# Summary

Orleans 2.0 is based on .NET Standard 2.0, so Orleans can now run on .NET Core and the full .NET Framework alike. It can be run on any platform capable of running .NET Core, Linux being just one example.

A big thanks goes to the Microsoft Orleans team for making this happen! (And to Richard Astbury for the awesome Orleans Dashboard, which he still claims is alpha quality.)

To recap: in order to have a minimal Orleans sample running, we need to:

1. Install the necessary packages.
2. Add a grain and a grain interface.
3. Set up the silo and client.
4. Use the grain from the client.
5. Optionally, set up logging and the Orleans Dashboard.

This example is meant to get you quickly up and running, and does not delve into any proper project structure or optimisations, which you would normally have when building a serious solution around Orleans. In the next Orleans 2.0 article, we’ll see how to properly organise an Orleans 2.0 solution.

<https://dotnet.github.io/orleans/Documentation/tutorials_and_samples/tutorial_1.html>