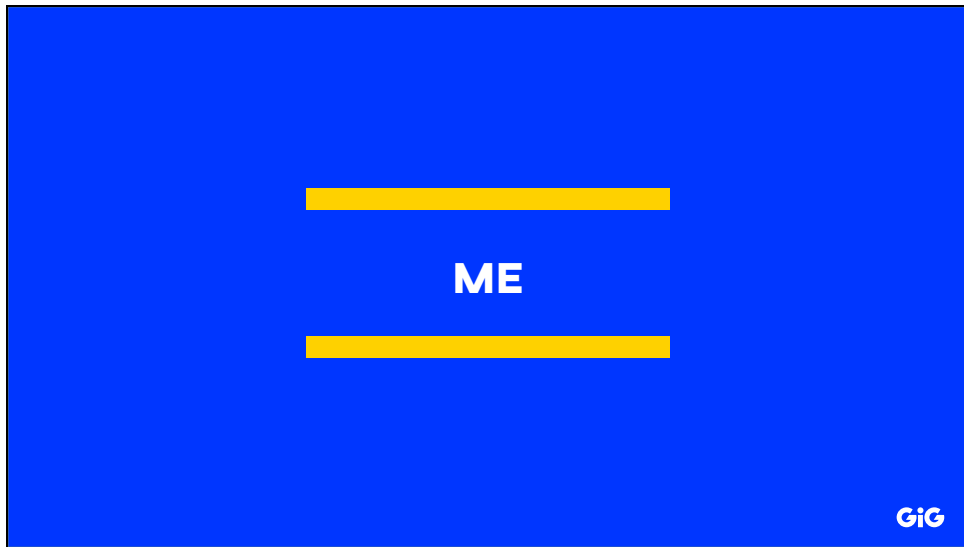


Slide 1



AGENDA

- Me
- GiG Norway – Who we are, what we do
- Actor Model
- Microsoft Orleans – What it is and how it works
- Hello World Demo
- System Overview - GiG Sports Connect (systems from Norway)
- Suggestions for monitoring (with demo)



ME

- From Kristiansand
- UiA and Queensland University of Technology (Australia) Master IT
- Work experience with Systems development since January 2001 (Oslo + Kristiansand)
- The last 4 years primarily worked on business- and development processes and architecture
- Bård Eik
- Epost: bardeik@outlook.com
- Twitter: [@bardeik1337](https://twitter.com/bardeik1337)



Slide 6





WHAT WE DO

2 primary lines of business

Betting operation

- Place bets on sports events (football)
- New sports to come:
 - Tennis, Ice hockey, Basket
- Place manual and automated (robot) bets in the Asian bookie market
- The Betting operation have two objectives
 - Generate revenue
 - Field testing of probability engines

Probability data feed

- Produce data feed containing betting markets and probability, e.g.:

```
<market id=123 name=1x2>
  <outcome name=home odds=1,79>
  <outcome name=draw odds=2,43>
  <outcome name=away odds=4,32>
</market>
or
<market id=123 name=1x2>
  <outcome name=home probability=64%>
  <outcome name=draw probability=27%>
  <outcome name=away probability=9%>
</market>
```

- This data can be used for various purposes. The current customers/consumers are:
 - Media (TV2)
 - Betting operations (GiG)

GiG



ACTOR MODEL

- The actor model adopts the philosophy that *everything is an actor*.
- The **actor model** in [computer science](#) is a [mathematical model](#) of [concurrent computation](#) that treats "actors" as the universal primitives of concurrent computation.
- In response to a [message](#) that it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received.
- Actors may modify their own [private state](#), but can only affect each other through messages (avoiding the need for any [locks](#)).
- The actor model originated in 1973.^[1]
- It has been used both as a framework for a [theoretical understanding](#) of [computation](#) and as the theoretical basis for several [practical implementations](#) of [concurrent systems](#).

GiG

https://en.wikipedia.org/wiki/Actor_model

Carl Hewitt; Peter Bishop; Richard Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.

Alt er en actor

En actor er det minste elementet i en modell hvor man kjører kommandoer i parallell

En actor kan sende meldinger, lage andre actorer, fortelle seg selv hvordan den neste meldingen skal håndteres

En actor kan endre på privat state, men ikke på andre actors, så den kan kun påvirke andre actors ved å sende dem meldinger

På denne måten unngår man å håndtere låser.

Actor modellen blir brukt i både teoretisk forskning og er implementert i rammeverk, som for eksempel Orleans

ACTOR FUNDAMENTAL CONCEPTS

- An actor is a computational entity that, in response to a message it receives, can concurrently:
 - send a finite number of messages to other actors;
 - create a finite number of new actors;
 - designate the behaviour to be used for the next message it receives.
- There is no assumed sequence to the above actions and they could be carried out in parallel.
- Decoupling the sender from communications sent was a fundamental advance of the Actor model enabling [asynchronous communication](#) and control structures as patterns of [passing messages](#).^[8]
- Recipients of messages are identified by address, sometimes called "mailing address". Thus an actor can only communicate with actors whose addresses it has. It can obtain those from a message it receives, or if the address is for an actor it has itself created.

GiG

Sende et bestemt antall meldinger

Lage et bestemt antall actors

Sette opp oppførsel for hvordan håndtere neste melding den mottar

Disse kan skje i hvilken som helst rekkefølge, og kjøre parallellt

Decoupling av avsender fra kommunikasjonsleddet var en fundamental endring slik at man fikk asynkron kommunikasjon og kontrollstrukturer som patterns på meldingsutveksling.

Mottakere har en adresse. Denne adressen kan enten være kjent for actoren eller den kan motta slike i meldinger.

En actor har adressen til alle den har opprettet selv.



WHAT IS ORLEANS?

- A Framework
- For building distributed high-scale computing applications
- No need to learn and apply complex concurrency or other scaling patterns
- Created for the Cloud from the start
- Created by Microsoft Research
- Open Source
- Virtual Actors
- Starting point for further info: <https://dotnet.github.io/orleans>

GiG


Microsoft Orleans er et rammeverk som tilbyr en relativt enkel fremgangsmåte for å bygge distribuerte og svært skalerbare programmer som kan håndtere store mengder data.

SOME COMPANIES USING ORLEANS

- [Gassumo](#)
- [Microsoft](#) Skype, Azure, others
- [Microsoft Studios](#) 343 Studios (*Halo*), Age of Empires, BigPark, Black Tusk, others
- [Microsoft Research](#)
- [NašeUkoly.CZ](#)
- [Trustev](#)
- [Mailcloud Limited](#)
- [Gigya](#)
- [Honeywell](#)
- [Mesh Systems](#) MESHVista Smart Cloud IoT Platform leverages Orleans for back-end services monitoring device state and business logic
- [Applicita Limited](#) A number of client projects where extreme scale and performance is required
- [Drawboard](#) Cloud collaboration and synchronisation platform
- [YouScan](#) Social media monitoring & analytics provider. Orleans is used for stateful stream processing at scale, reliable execution of long running jobs and as a main application server.
- [Visa](#)
- [PagoLivre](#) Mobile and Social Payment platform
- [invertirOnline.com](#) Argentinian-based electronic brokerage firm
- [Lebara](#)
- [Nomnio](#) Nomnio IoT Platform and industry projects where reliability and performance is required
- [Real Artists Ship 2.0](#) Fast, native, comprehensive issue tracking for GitHub

SOME PROJECTS AND APPLICATIONS

- [Halo 4 - 343 Industries](#)
- [Microsoft BigPark Studio](#)
- [Orleans-Contrib](#) Community contribution projects, including Orleans Monitoring, Design Patterns, Storage Provider, ...
- [Pegasus Mission](#) More Info [here](#) and [here](#)
- [Sync.Today 2015](#) .NET Business Processes Automation Platform. More info [here](#)
- [Microdot](#) A microservices framework by Gigya, for writing Orleans based microservices




KEY SELLING POINTS

Scalable by Default
Orleans handles the complexity of building distributed systems, enabling your application to scale to hundreds of servers.

Low Latency
Orleans allows you to keep the state you need in memory, so your application can rapidly respond to incoming requests.


Simplified Concurrency
Orleans allows you to write simple, single threaded C# code, handling concurrency with asynchronous message passing between objects (grains).



Skalerbart, raskt, håndterer rekkefølgen på oppgaver som skal utføres for deg.

BENEFITS

- Developer Productivity
 - Familiar object-oriented programming (OOP) paradigm
 - Single-threaded execution of actors
 - Transparent activation
 - Location transparency
 - Transparent integration with persistent store
 - Automatic propagation of errors
- Transparent Scalability by Default
 - Implicit fine grain partitioning of application state
 - Adaptive resource management
 - Multiplexed communication
 - Efficient scheduling
 - Explicit asynchrony



Transparent integration with persistent store – garanterer lagring er utført når klienten får svar.

Multiplexed communication. – Logiske endpoints og kjører meldinger mellom dem på en all-to-all fysisk connection (TCP socket). Aktivering og deaktivering av Actor involverer ikke fysiske endpoints, så det går veldig raskt uten kost på dette.

Efficient scheduling. – Runtimeen skedulerer et stort antall single threaded actors på tvers av et custom thread pool med en tråd per fysiske prosessor kjerne. Ved å skrive kode som ikke blokkerer (noe Orleans per definisjon krever), kjører koden veldig effektivt. Dette fører til CPU utilization opp til 90%+

Explicit asynchrony. Programmeringsmodellen I Orleans guider programmerere til å skrive kode som ikke blokkerer og er asyncon.

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

Benefits

The main benefits of Orleans are: **developer productivity**, even for non-expert programmers; and **transparent scalability by default** with no special effort from the programmer. We expand on each of these benefits below.

Developer Productivity

The Orleans programming model raises productivity of both expert and non-expert programmers by providing the following key abstractions, guarantees and system services.

Familiar object-oriented programming (OOP) paradigm. Actors are .NET classes that implement declared .NET actor interfaces with asynchronous methods. Thus actors appear to the programmer as remote objects whose methods can be directly invoked. This provides the programmer the familiar OOP paradigm by turning method calls into messages, routing them to the right endpoints, invoking the target actor's methods and dealing with failures and corner cases in a completely transparent way.

Single-threaded execution of actors. The runtime guarantees that an actor never executes on more than one thread at a time. Combined with the isolation from other actors, the programmer never faces concurrency at the actor level, and hence never needs to use locks or other synchronization mechanisms to control access to shared data. This feature alone makes development of distributed applications tractable for non-expert programmers.

Transparent activation. The runtime activates an actor as-needed, only when there is a message for it to process. This cleanly separates the notion of creating a reference to an actor, which is visible to and controlled by application code, and physical activation of the actor in memory, which is transparent to the application. In many ways, this is similar to virtual memory in that it decides when to "page out" (deactivate) or "page in" (activate) an actor; the application has uninterrupted access to the full "memory space" of logically created actors, whether or not they are in the physical memory at any particular point in time. Transparent activation enables dynamic, adaptive load balancing via placement and migration of actors across the pool of hardware resources. This feature is a significant improvement on the traditional actor model, in which actor lifetime is application-managed.

Location transparency. An actor reference (proxy object) that the programmer uses to invoke the actor's methods or pass to other components only contains the logical identity of the actor. The translation of the actor's logical identity to its physical location and the corresponding routing of messages are done transparently by the Orleans runtime. Application code communicates with actors oblivious to their physical location, which may change over time due to failures or resource management, or because an actor is deactivated at the time it is called.

Transparent integration with persistent store. Orleans allows for declarative mapping of actors' in-memory state to persistent store. It synchronizes updates, transparently guaranteeing that callers receive results only after the persistent state has been successfully updated. Extending and/or customizing the set of existing persistent storage providers available is straight-forward.

Automatic propagation of errors. The runtime automatically propagates unhandled errors up the call chain with the semantics of asynchronous and distributed try/catch. As a result, errors do not get lost within an application. This allows the programmer to put error handling logic at the appropriate places, without the tedious work of manually propagating errors at each level.

Transparent Scalability by Default

The Orleans programming model is designed to guide the programmer down a path of likely success in scaling their application or service through several orders of magnitude. This is done by incorporating the proven best practices and patterns, and providing an efficient implementation of the lower level system functionality. Here are some key factors that enable scalability and performance.

Implicit fine grain partitioning of application state. By using actors as directly addressable entities, the programmer implicitly breaks down the overall state of their application. While the Orleans programming model does not prescribe how big or small an actor should be, in most cases it makes sense to have a relative large number of actors – millions or more – with each representing a natural entity of the application, such as a user account, a purchase order, etc. With actors being individually addressable and their physical location abstracted away by the runtime, Orleans has enormous flexibility in balancing load and dealing with hot spots in a transparent and generic way without any thought from the application developer.

Adaptive resource management. With actors making no assumption about locality of other actors they interact with and because of the location transparency, the runtime can manage and adjust allocation of available HW resources in a very dynamic way by making fine grain decisions on placement/migration of actors across the compute cluster in reaction to load and communication patterns without failing incoming requests. By creating multiple replicas of a particular actor the runtime can increase throughput of the actor if necessary without making any changes to the application code.

Multiplexed communication. Actors in Orleans have logical endpoints, and messaging between them is multiplexed across a fixed set of all-to-all physical connections (TCP sockets). This allows the runtime to host a very large number (millions) of addressable entities with low OS overhead per actor. In addition, activation/deactivation of an actor does not incur the cost of registering/unregistering of a physical endpoint, such as a TCP port or a HTTP URL, or even closing a TCP connection.

Efficient scheduling. The runtime schedules execution of a large number of single-threaded actors across a custom thread pool with a thread per physical processor core. With actor code written in the non-blocking continuation based style (a requirement of the Orleans programming model) application code runs in a very efficient “cooperative” multi-threaded manner with no contention. This allows the system to reach high throughput and run at very high CPU utilization (up to 90%+) with great stability. The fact that a growth in the number of actors in the system and the load does not lead to additional threads or other OS primitives helps scalability of individual nodes and the whole system.

Explicit asynchrony. The Orleans programming model makes the asynchronous nature of a distributed application explicit and guides programmers to write non-blocking asynchronous code. Combined with asynchronous messaging and efficient scheduling, this enables a large degree of distributed parallelism and overall throughput without the explicit use of multi-threading.

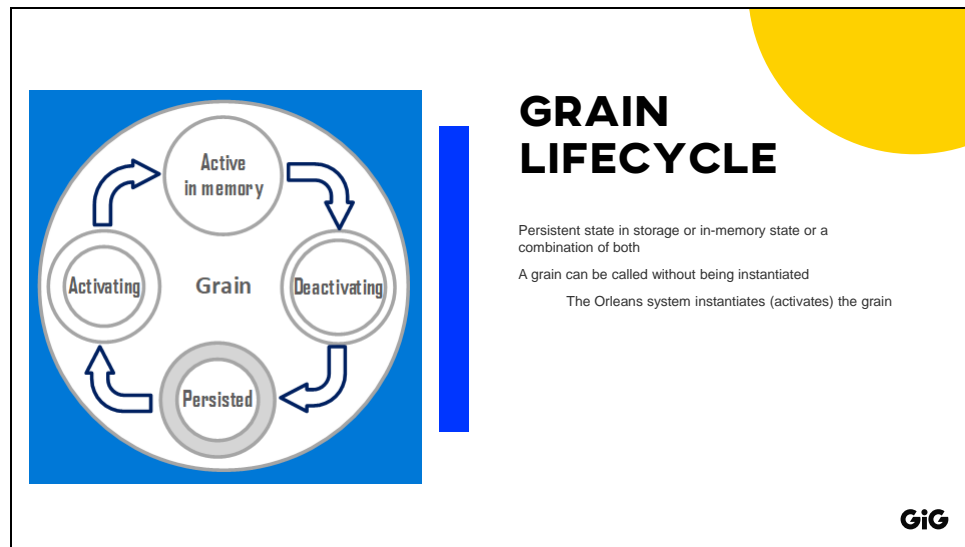
VIRTUAL ACTORS

- The Actor Model has been around since the 1970's
- Traditional actors in for instance Erlang and Akka
- Orleans Grains are Virtual Actors
 - Which means that physical instantiations of grains are completely abstracted away and are automatically managed by the Orleans runtime
 - Much more suitable for high-scale dynamic workloads like cloud services and is the major innovation of Orleans
 - You can read more details in the [Technical Report on Orleans](#).

GiG

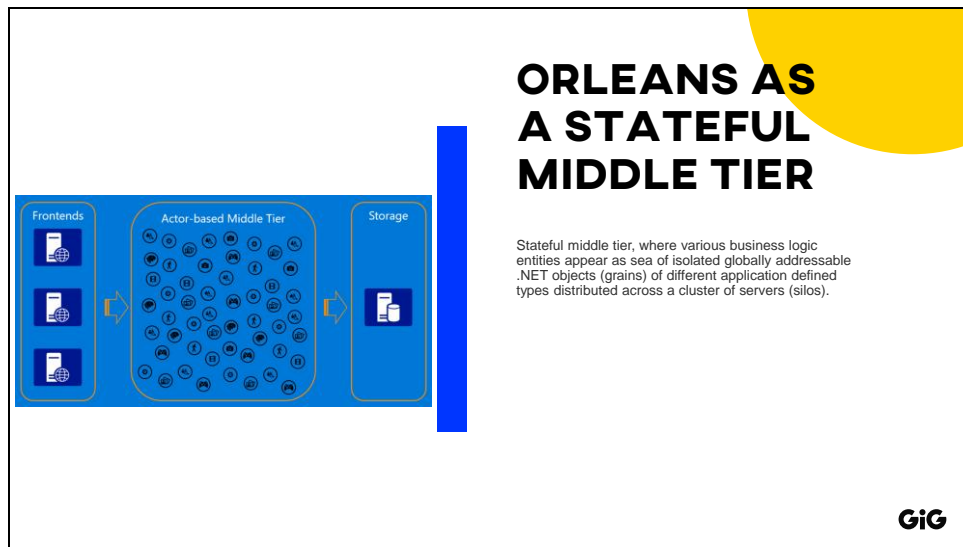
Virtual Actor modellen er meget egnet for høy skalerbarhet og dynamisk håndtere last i en skyverden, og er den største innovasjonen i Orleans.

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](#) 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. You can read more details in the [Technical Report on Orleans](#).

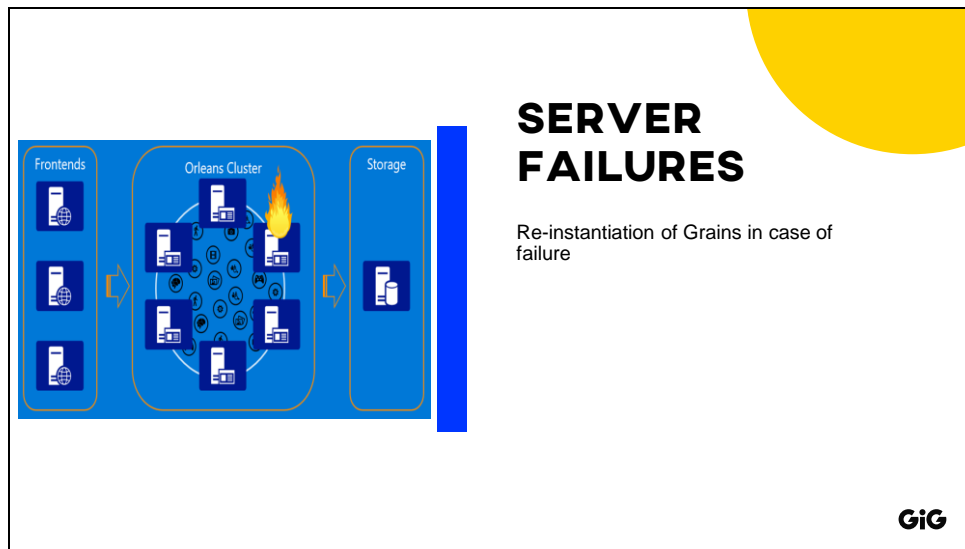


Grain can have persistent state in storage or in-memory state or a combination of both. Any grain can be called by any other grain or by a frontend (client) by using the target grain's logical identity without the need to ever create or instantiate the target grain. The Orleans programming model makes grains appear as if they are in memory the whole time. In reality, a grain goes through the lifecycle from existing only as its persisted state in storage to being instantiated in memory to being removed from memory.

The Orleans runtime behind the scene instantiates (activates) grains when there's work for them to do, and removes them from memory (deactivates) to reclaim hardware resources when they are idle for too long. This grain lifecycle management work of the runtime is transparent to the application code, and liberates it from the complicated task of distributed resource management. Application logic can be written with the whole "address space" of grains available to it without the need to have hardware resources to keep all grains in memory at the same time, conceptually similar to how virtual memory works in operating systems.



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).



The virtual nature of grains allows Orleans to handle server failures mostly transparently to the application logic because grains that were executing on a failed server get automatically re-instantiated on other servers in the cluster once the failure is detected.



INTERFACE

```
public interface IMyGrain : IGrainWithStringKey
{
    Task<string> SayHello(string name);
}
```

GiG

In Orleans, grains are the building blocks of application code. Grains are instances of .NET classes that implement a conforming interface. Asynchronous methods of the interface are used to indicate which operations the grain can perform

IMPLEMENTATION

```
public class MyGrain : IMyGrain
{
    public Task<string> SayHello(string name)
    {
        return Task.FromResult($"Hello {name}");
    }
}
```

GiG

The implementation is executed inside the Orleans framework:

INVOKING A PROXY OBJECT

```
var grain = GrainClient.GrainFactory.GetGrain<IMyGrain>("grain1");  
await grain.SayHello("World");
```

GiG

You can then invoke the grain by obtaining a proxy object (a grain reference), and calling the methods:

HELLO WORLD

- Based on: <https://github.com/dotnet/orleans/tree/master/Samples/2.0/HelloWorld>
- Added persistence for messages and some client logic, using this.
- Hello World Demo
 - To debug, set both as startup projects in solution
- To start the silo
 - `dotnet run --project src\SiloHost`
- To start the client (you will have to use a different command window)
 - `dotnet run --project src\OrleansClient`



Cluster of silo type A | Cluster of Silotype B | Cluster of Silotype C | Cluster of Silotype D

Original thoughts: Observer Pattern from right to left

Challenge: ClusterClientCommunication could take up to 10 seconds to get connected

Ended up with:

One cluster of 3 silos of the same type

Subscribe to changes in State Controller (Matches, Players, other stats)

One Way attribute (fire and forget)

OOPS:

Default Orleans communication/messaging style is at most once -> Needs to transfer to at least once

Default Timeout for operations are 30 seconds

If a task runs for more than 200 ms, Orleans will warn that the task takes longer than expected



SOME OF OUR SYSTEM MONITORING

- Service Fabric Explorer – services
- Grafana - servers
- Orleans Dashboard - Orleans

