Microsoft Orleans in Betting

Andrii Laptiuk
.NET developer @ Platform Team

BETLĀB

Agenda

- Motivation
- What is Orleans?
- What about betting?
- Programming model in examples
- What about scalability?
- References

BETLĀB

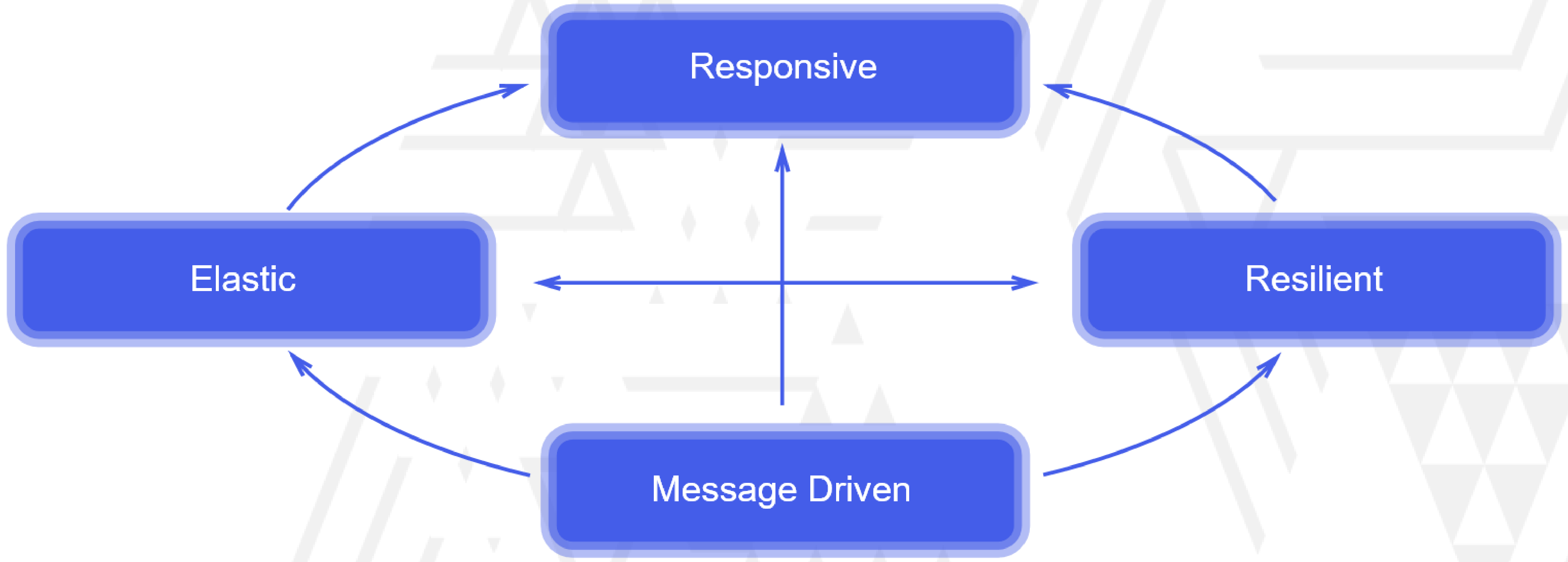
Motivation

Building real-time analytics system for monitoring dynamic information about betting events progress with further feedback to upstream systems

BETLĀB

The Reactive Manifesto

- Responsive
- Resilient
- Elastic
- Message Driven



BETLĀB

What is Orleans?

- “Orleans is a framework that provides a straightforward approach to building distributed high-scale computing applications, without the need to learn and apply complex concurrency or other scaling patterns”
- “Novel abstraction of virtual actors that solves a number of the complex distributed systems problems”

Microsoft

BETLĀB

What is Orleans?

- Simplified Programming Model
- Clever Distributed Runtime
- Scalable by design

BETLĀB

Simplified Programming Model & Runtime

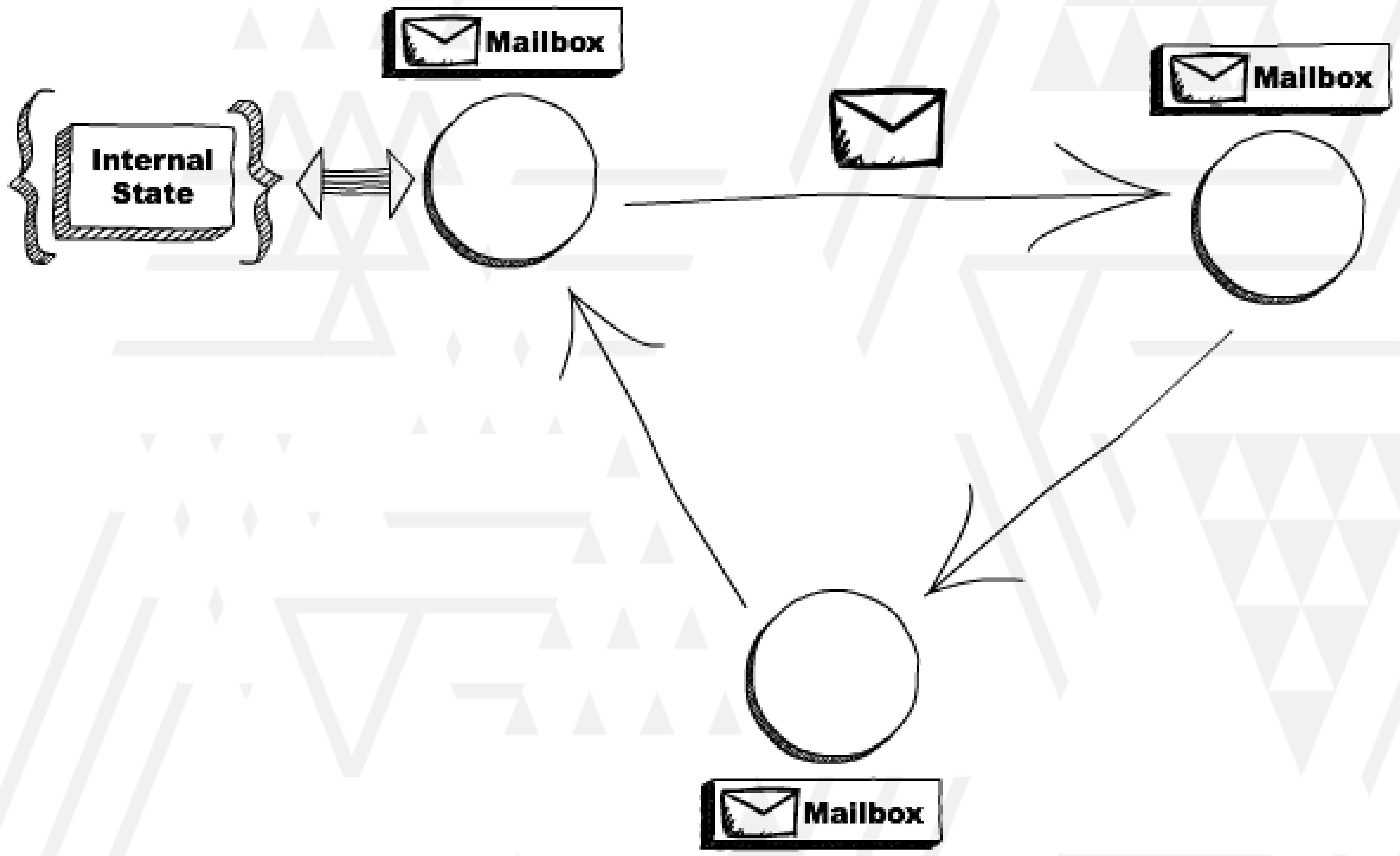
BETLAB

Object-oriented programming

- Everything is an object
- Communicate by sending and receiving messages
- Objects have their own memory
- Every object is an instance of a class
- The class holds the shared behavior for its instances

Actor model

- Everything is an actor
- Isolated state
- Asynchronous message passing communication
- Mailbox
- Single-threaded execution model
- Addressable unit



BETLĀB

Responsibility on developer

- Inherent complexity of distributed systems domain
- Minimal abstraction to provide without compromises
- Manage lifecycle of actors
- Handle failures and recovery of actors
- Distributed resource management

BETLĀB

Orleans: Cloud Computing for Everyone

- Simplify distributed computing and allow non-experts to write efficient, scalable and reliable distributed services
- Default behavior that is most natural and easy for non-experts
- Automatic resource management
- Automatic actor placement and load balancing

BETLĀB

Virtual actors

- Perpetual existence
- Automatic instantiation
- Location transparency
- Automatic scale-out

Back to betting

Sports events

BETLAB

Match in progress…

BETLAB

Players

BETLĀB

Bookmakers

Markets

BETLĀB

Player responsibilities

- Place bet

Event responsibilities

- Handle event state changes
- Handle scoreboard changes

BETLĀB

Market responsibilities

- Process bet
- Handle odds changes
- Deactivation

BETLĀB

Back to code

BETLĀB

Introducing abstractions

```
public interface IPlayer
{
    void PlaceBet(Bet bet);
}

public interface IEvent
{
    void SetEventInformation(EventInformation eventInformation);
    void SetScoreboard(ScoreboardData scoreboardData);
    InnerFeedEventData GetSnapshot();
}

public interface IMarket
{
    void ReceiveBet(Bet bet);

    void ChangeOdds(OddData[] odds);

    void Deactivate();
}
```

BETLĀB

Grain contracts

```
Grain identity:
    long, string, Guid

Grain interfaces:
    IGrainWithIntegerKey, IGrainWithStringKey, IGrainWithGuidKey

Grain contract constraints:
    Every method should return Task or Task<T>
```

```
public interface IMarket : IGrainWithStringKey
{
    Task ReceiveBet(Bet bet);

    Task Deactivate();

    Task ChangeOdds(OddData[] odds);
}
```

BETLĀB

Grains

Base class:

    Grain

Execution model:

    Single-threaded via custom single-thread task scheduler

Grain constraints:

    Method implementation should not contain blocking calls

```
public class MarketActor : Grain, IMarket
{
      private OddData[] _odds;
      private bool _isActive;
      private int _betsCount;
      private decimal _totalAmount;

      public Task ReceiveBet(Bet bet)
      {
         _betsCount++;
         _totalAmount += bet.Amount;
         return TaskDone.Done;
      }

      public Task Deactivate()
      {
         _isActive = false;
         return TaskDone.Done;
      }

      public Task ChangeOdds(OddData[] odds)
      {
         _isActive = true;
         _odds = odds;
         return TaskDone.Done;
      }
}
```

BETLĀB

Declarative persistence

Base class:
Grain<T>

Main abstraction:
IStorageProvider

Configuration:
Both on application level and per grain type, in code or in config file

```
[StorageProvider(ProviderName = "RedisStorageProvider")]
public class MarketActor : Grain<MarketState>, IMarket
{
    public Task ReceiveBet(Bet bet)
    {
        State.BetsCount++;
        State.TotalAmount += bet.Amount;
        return WriteStateAsync();
    }

    public Task Deactivate()
    {
        State.IsActive = false;
        return WriteStateAsync();
    }

    public Task ChangeOdds(OddData[] odds)
    {
        State.IsActive = true;
        State.Odds = odds;
        return WriteStateAsync();
    }
}
```

BETLAB

Persistent state

Constraints:

State object should be serializable

```
[Serializable]
public class MarketState
{
    public OddData[] Odds { get; set; }
    public bool IsActive { get; set; }
    public int BetsCount { get; set; }
    public decimal TotalAmount { get; set; }
}
```

Journaled grains

Base class:

```
JournaledGrain<TGrainState, TEventBase>
```

Main abstraction:

```
LogConsistencyProvider
```

Constraints:

All state and event objects should be serializable

```
[LogConsistencyProvider(ProviderName = "LogStorage")]
public class MarketActor : JournaledGrain<MarketState, IMarketEvent>, IMarket
{
    public async Task ReceiveBet(Bet bet)
    {
        RaiseEvent(new Bet(bet.Amount));
        await ConfirmEvents();
    }

    public async Task Deactivate()
    {
        RaiseEvent(new MarketDeactivated());
        await ConfirmEvents();
    }

    public async Task ChangeOdds(OddData[] odds)
    {
        RaiseEvent(new MarketOddsChanged{Odds = odds});
        await ConfirmEvents();
    }
}
```

BETLĀB

Events

```
public interface IMarketEvent
{
}

[Serializable]
public class Bet : IMarketEvent
{
    public decimal Amount { get; set; }
}

[Serializable]
public class MarketDeactivated : IMarketEvent
{
}

[Serializable]
public class MarketOddsChanged : IMarketEvent
{
    public OddData[] Odds { get; set; }
}
```

BETLĀB

State transitions

```
[Serializable]
public class MarketState
{
    public OddData[] Odds { get; set; }
    public bool IsActive { get; set; }
    public int BetsCount { get; set; }
    public decimal TotalAmount { get; set; }

    public void Apply(Bet @event)
    {
        BetsCount++;
        TotalAmount += @event.Amount;
    }

    public void Apply(MarketDeactivated @event)
    {
        IsActive = false;
    }

    public void Apply(MarketOddsChanged @event)
    {
        IsActive = true;
        Odds = @event.Odds;
    }
}
```

BETLĀB

Server-side interaction

Get reference with calling:

```
GrainFactory.GetGrain<IGrainInterface>(key)

public class PlayerActor : Grain, IPlayer
{
    public async Task PlaceBet(Bet bet)
    {
        var marketActor = GrainFactory.GetGrain<IMarket>("[2,0,[1],0]");
        await marketActor.ReceiveBet(bet);
    }
}
```

BETLĀB

Client-side interaction

Get reference with calling:

```
    GrainClient.GrainFactory.GetGrain<IGrainInterface>(key)
```

```
var playerId = message.Chat.Id;
var playerActor = GrainClient.GrainFactory.GetGrain<IPlayer>(playerId);
await playerActor.PlaceBet(new Bet {Amount = 10 });
```

BETLĀB

Silo

Main function:
Hosts and executes Orleans grains

Communication:
Listening one port for silo-to-silo messaging
and another for client-to-silo messaging

Silo identity: **ip:port:epoch**

BETLĀB

Back to scalability

Cluster

Definition:

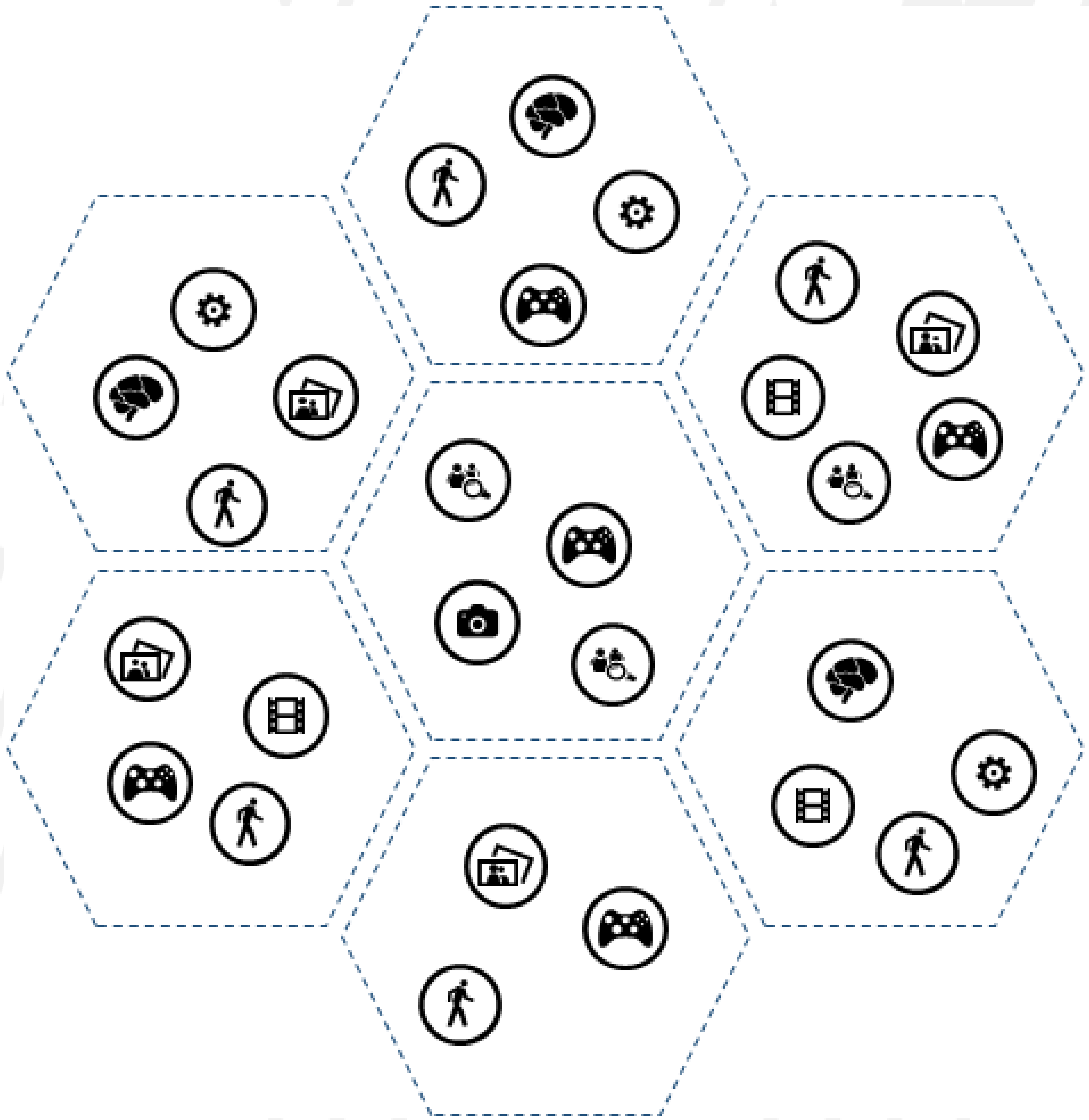
Number of silos working together

Cluster management:

Orleans runtime provides full automation

Implementation:

Shared membership store which updates automatically



Cluster membership

Implemented via:

Built-in membership protocol

Abstraction:

Protocol relies on external service with `MembershipTable` abstraction

Constraints:

Silo identity(**ip:port:epoch**) should be unique in deployment

Main functions:

- Detect failed silos
- Reach agreement on set of alive silos
- Handle new joins to cluster

Transparent scalability by default

- Application state distribution + load balancing
- Adaptive resource management
- Multiplexed communication
- Efficient scheduling
- Explicit asynchrony

BETLĀB

Deployment

- On-Premise
- Cloud

BETLĀB

Monitoring

Runtime tables:

- Silo Instances table
- Reminders table
- Silo Metrics table
- Clients Metrics table
- Silo Statistics table
- Clients Statistics table

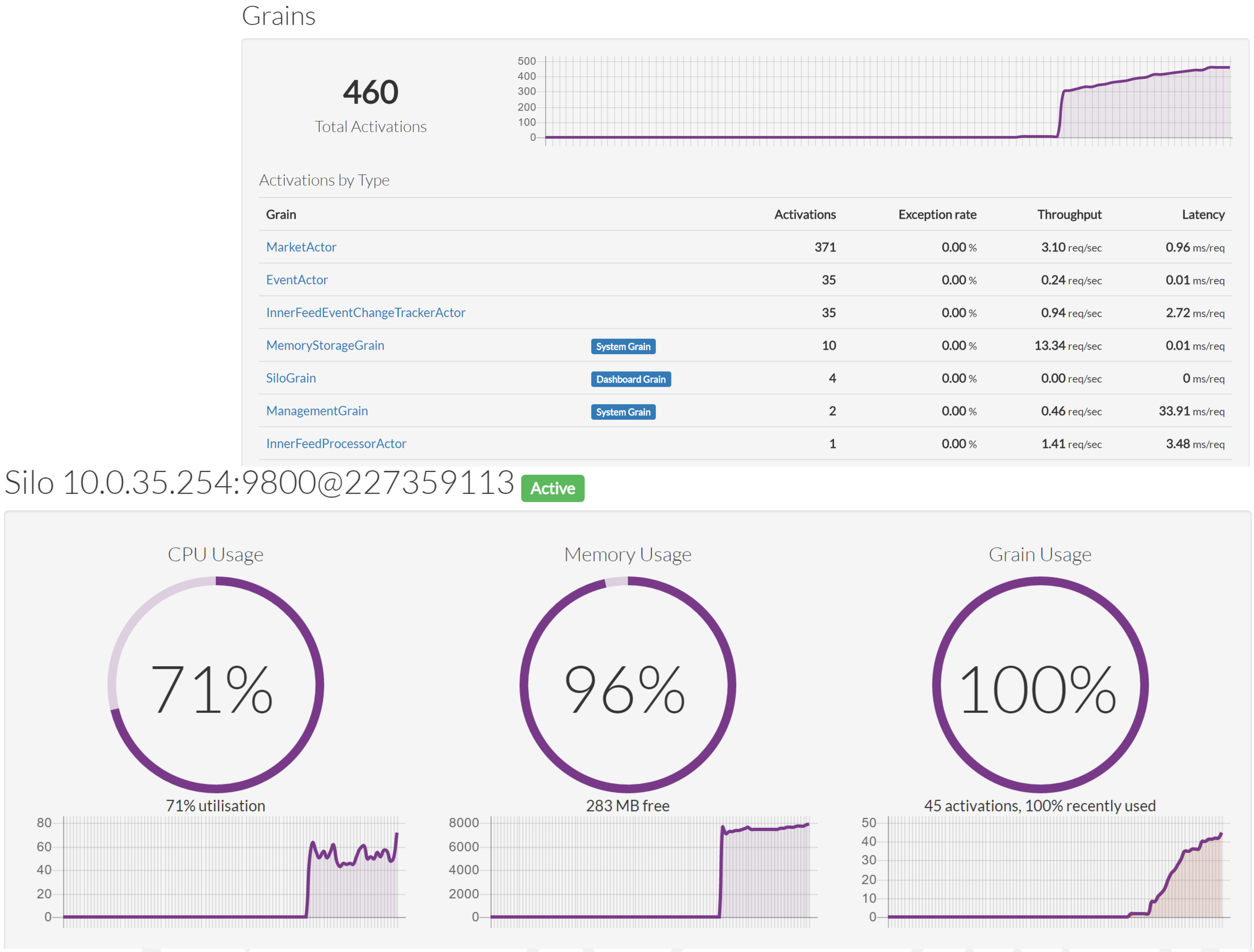
Orleans trace logging:

Telemetry consumers

Orleans dashboard:

NuGet package from Orleans community

OrleansDashboard



Grain		Activations	Exception rate	Throughput	Latency
MarketActor		371	0.00 %	3.10 req/sec	0.96 ms/req
EventActor		35	0.00 %	0.24 req/sec	0.01 ms/req
InnerFeedEventChangeTrackerActor		35	0.00 %	0.94 req/sec	2.72 ms/req
MemoryStorageGrain	System Grain	10	0.00 %	13.34 req/sec	0.01 ms/req
SiloGrain	Dashboard Grain	4	0.00 %	0.00 req/sec	0 ms/req
ManagementGrain	System Grain	2	0.00 %	0.46 req/sec	33.91 ms/req
InnerFeedProcessorActor		1	0.00 %	1.41 req/sec	3.48 ms/req

BETLĀB

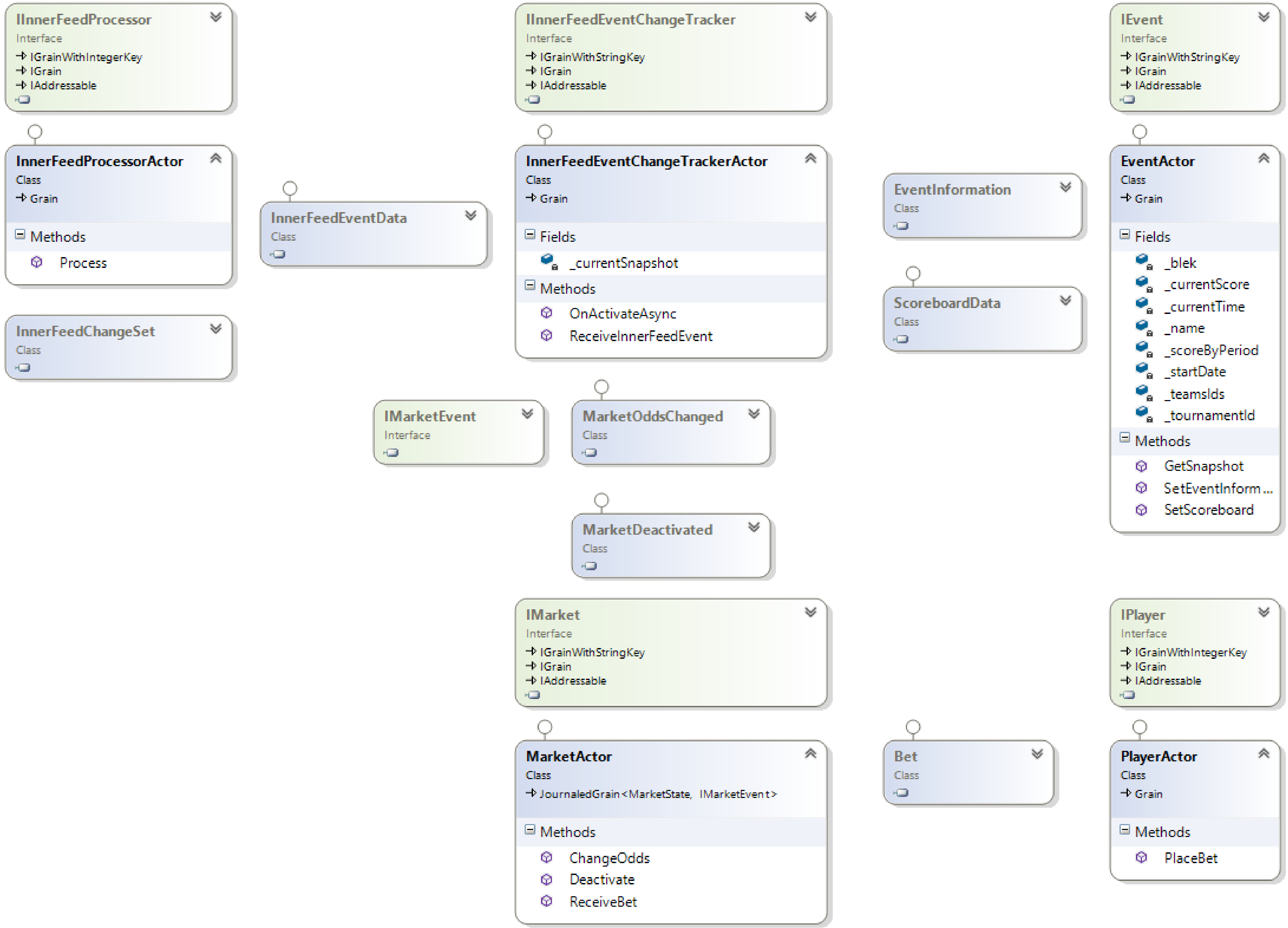
Demo

Telegram bot: @Net_Meetup_Bot

Source code:

https://github.com/applicativex/Meetup.Betting

Design



BETLĀB

Scenarios

- Social graphs
- Mobile backend
- Internet of things
- Real-time analytics
- ‘Intelligent’ cache
- Interactive entertainment

BETLĀB

Dangers

- Prisoner of Runtime
- Lack of documentation

BETLĀB

References

BETLĀB

References

- Documentation:
  http://dotnet.github.io/orleans
- OrleansContrib (community developed useful stuff):
  https://github.com/OrleansContrib/
- Useful for beginners list of Orleans design patterns:
  https://github.com/OrleansContrib/DesignPatterns
- Virtual meetups:
  https://www.youtube.com/results?search_query=Orleans+Virtual+Meetup
- Orleans vs Akka comparison:
  https://github.com/akka/akka-meta/blob/master/ComparisonWithOrleans.md/

BETLĀB

Questions

BETLĀB