

НАЗАД В БУДУЩЕЕ

Построение эффективных облачных сервисов
с помощью Orleans

Сергей Быков

Microsoft

@sergeybykov

Обо мене



OPT/MA

В Microsoft с 2001 г.

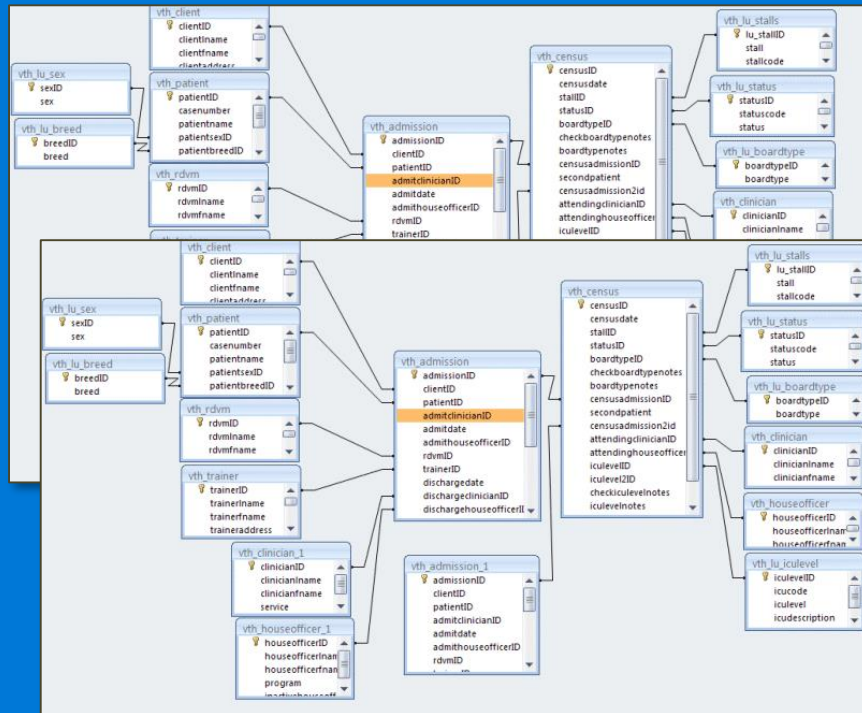
- Host Integration Server
- BizTalk Server
- Windows Embedded
- Bing
- Research
- Halo
- Xbox

В старые добрые *stateful* времена

Клиенты



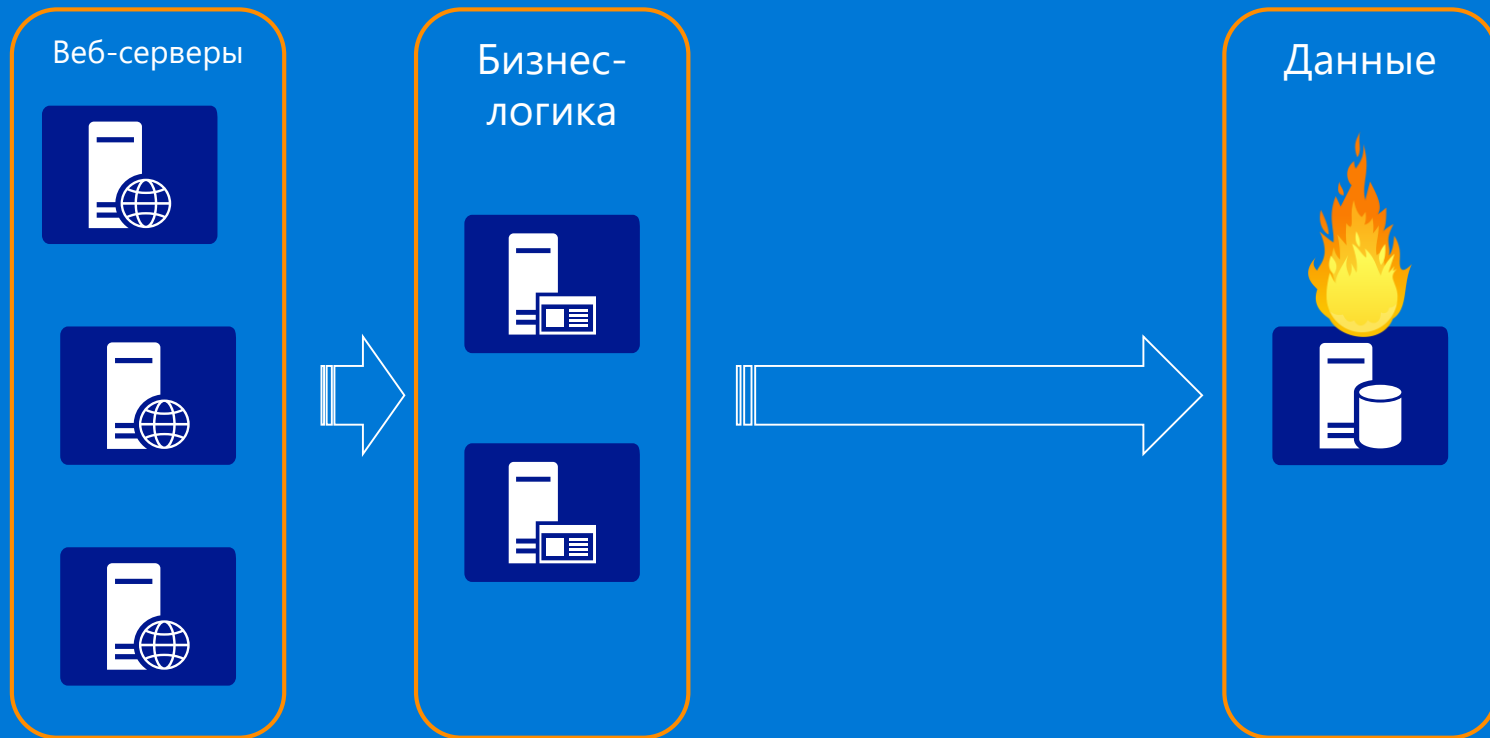
Сервер



Плюсы и минусы *stateful* модели

- + Низкая задержка
 - + Эффективный доступ к данным
 - + Координация стандартными примитивами
 - + Поддержка ООП
-
- Ограничена общей памятью
 - Проблема с масштабируемостью
 - Чувствительность к сбоям

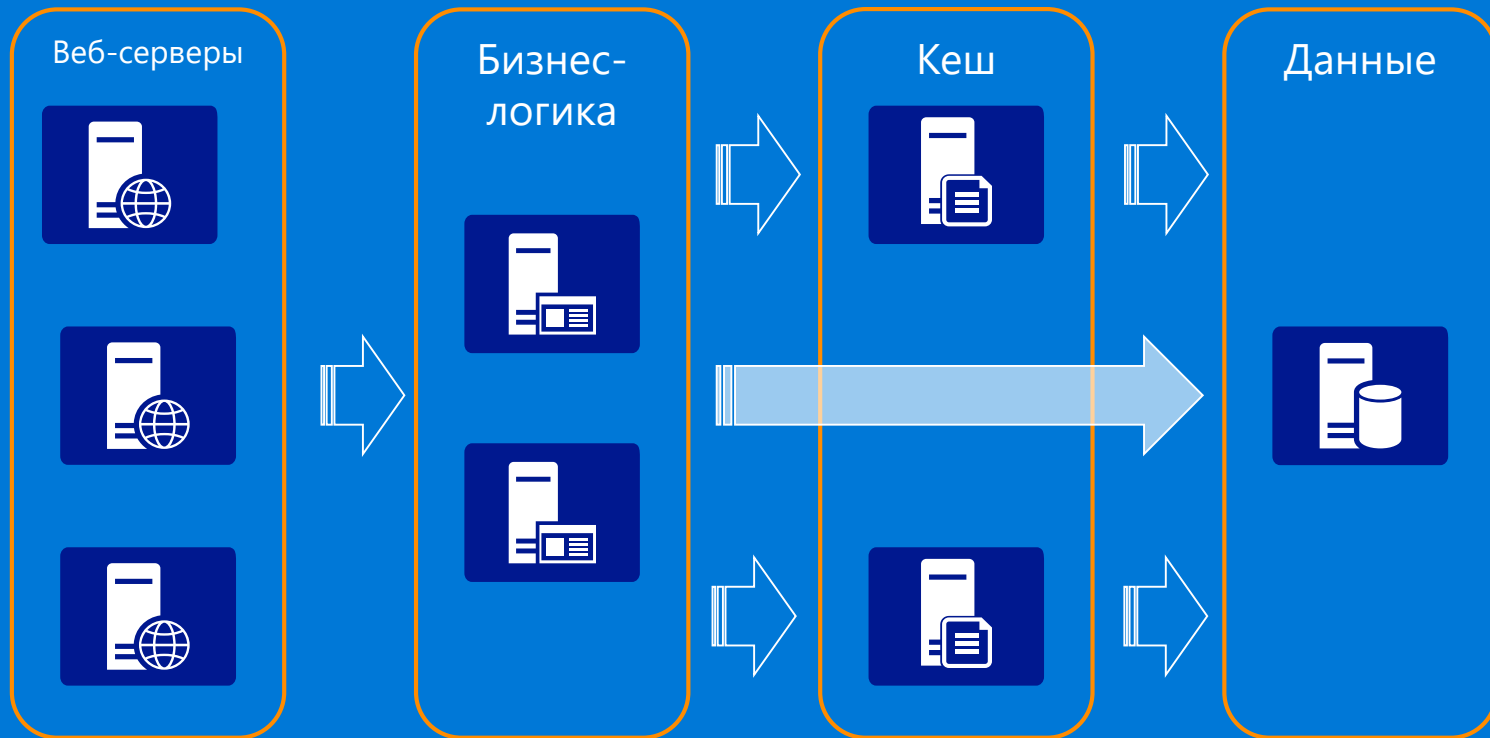
Stateless архитектура



Плюсы и минусы *stateless* модели

- + Простота масштабирования
- + Простота восстановления после сбоя
- Большая задержка
- Неэффективный доступ к данным
- Отсутствие координации
- Перенос сложности и нагрузки в слой данных
- Забываем про ООП

Stateless архитектура с кешем

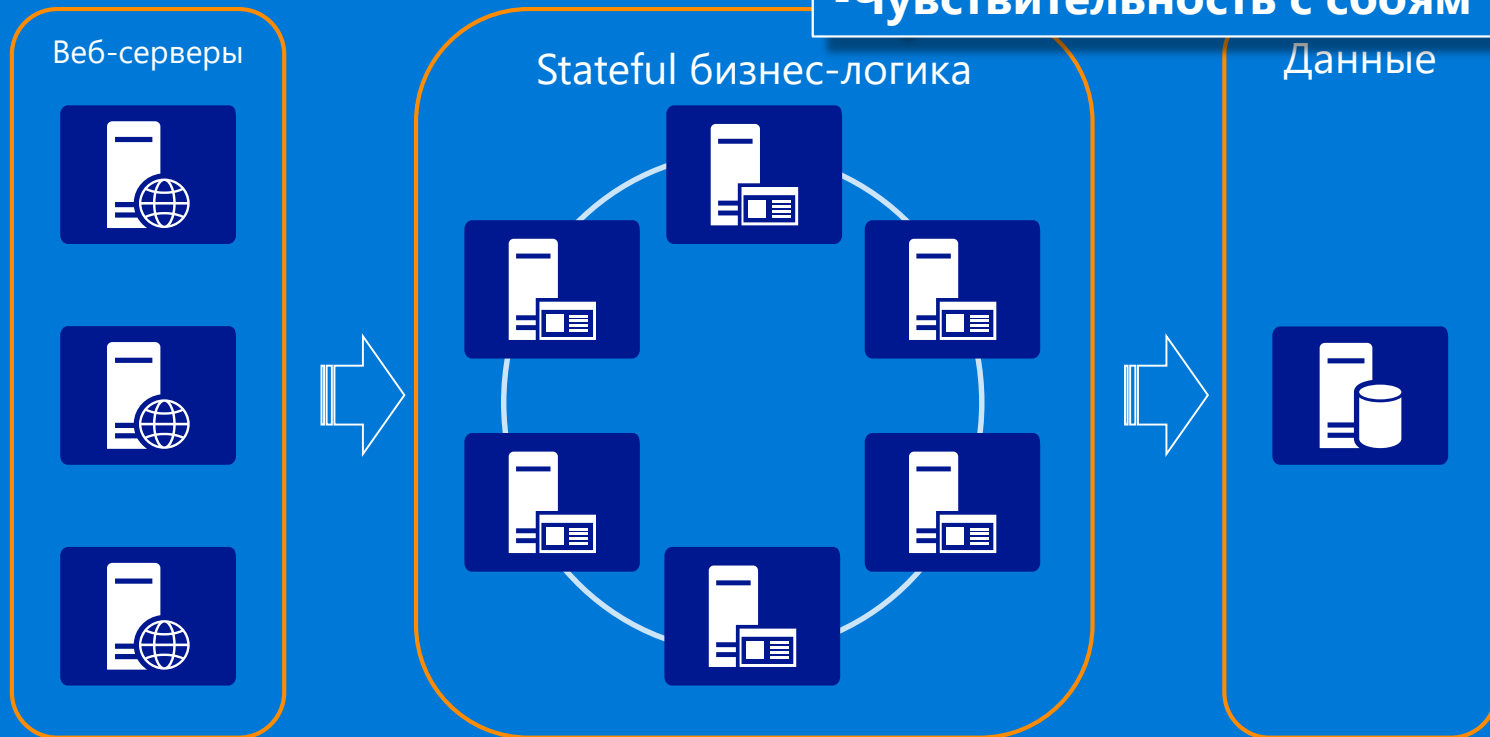


Плюсы и минусы *stateless* модели с кешем

- + Простота масштабирования
- + Простота восстановления после сбоя
- ~~— Большая задержка~~
- ~~— Неэффективный доступ к данным~~
- Отсутствие координации
- Перенос сложности и нагрузки в слой данных
- Забываем про ООП
- *Проблема инвалидации кеша*

Назад в *Stateful* будущее

- Ограничена общей памятью
- Проблема с масштабируемостью
- Чувствительность с сбоям



Orleans: Фреймворк для облаков*

Модель программирования

Для всех

Простая, но мощная

3x –10x меньше кода

Лёгкость масштабирования

Нет узких мест

Нет единой точки отказа

Объединяет ряд проверенных
методик распределённых систем

Суп из независимых объектов



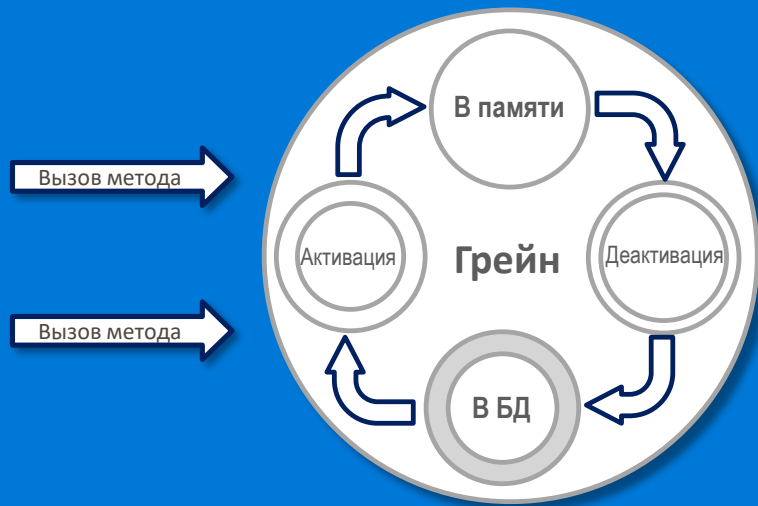
```
// grain interface
public interface IUser : IGrainWithGuidKey
{
    Task<string> SayHello(string name);
}
```

```
// invoking a grain
IUser user = client.GetGrain<IUser>(userId);
string reply = await user.SayHello(name);
Console.WriteLine($"User {userId} said: {reply}");
```

```
// grain implementation class
public class UserGrain : Grain, IUser
{
    private int _counter;

    public async Task<string> SayHello(string name)
    {
        return $"Hello, {name}. You are caller #{++_counter}.";
    }
}
```

Вызовы всегда
ОДНОПОТОЧНЫ

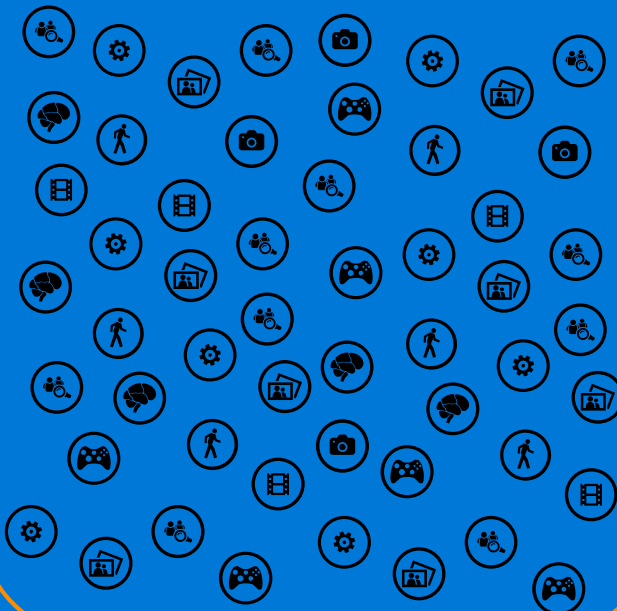


Скользящее подмножество всех
возможных объектов: пользователи,
устройства, сессии и т.д.

Веб-серверы



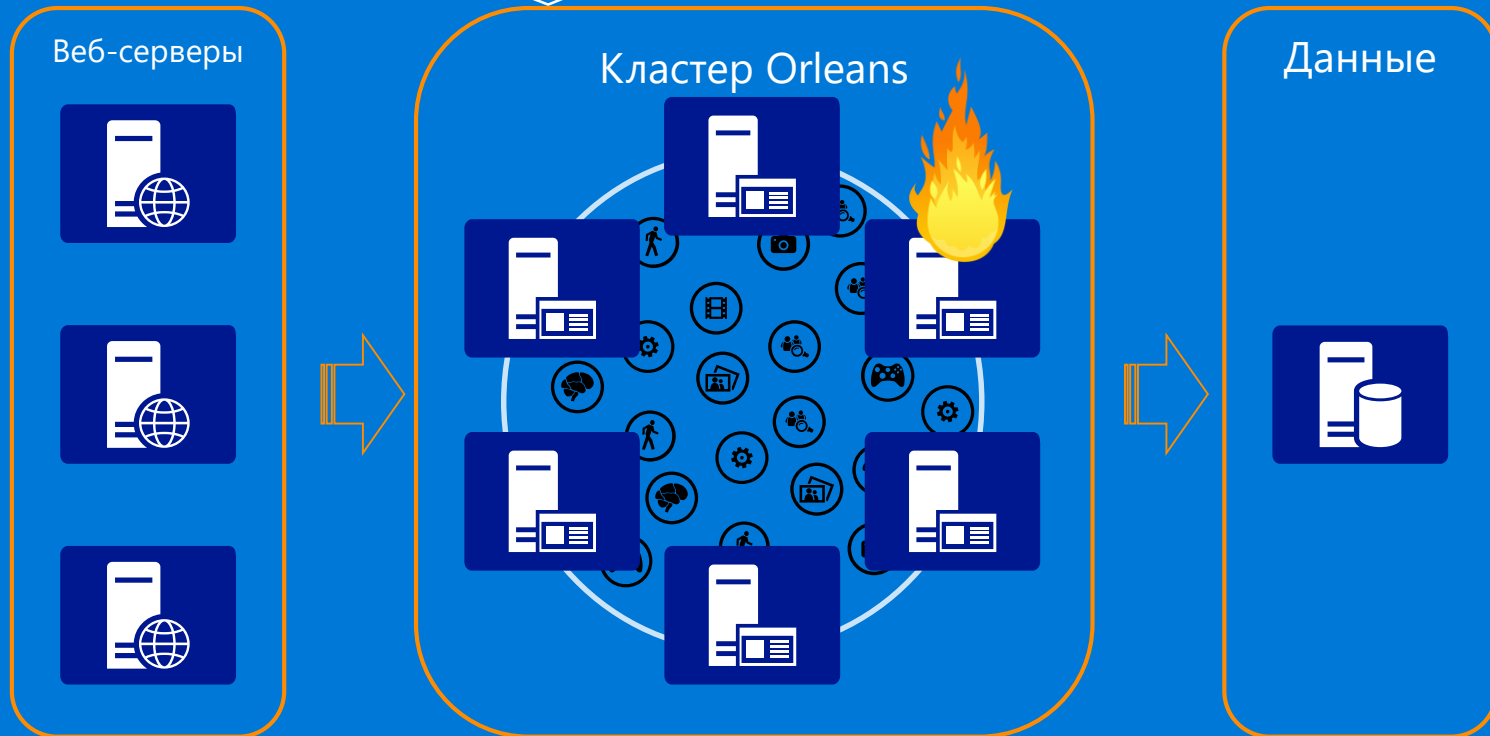
Бизнес-логика



Данные



Orleans прозрачно восстанавливается
после сбоев серверов и
переконфигурирует кластер



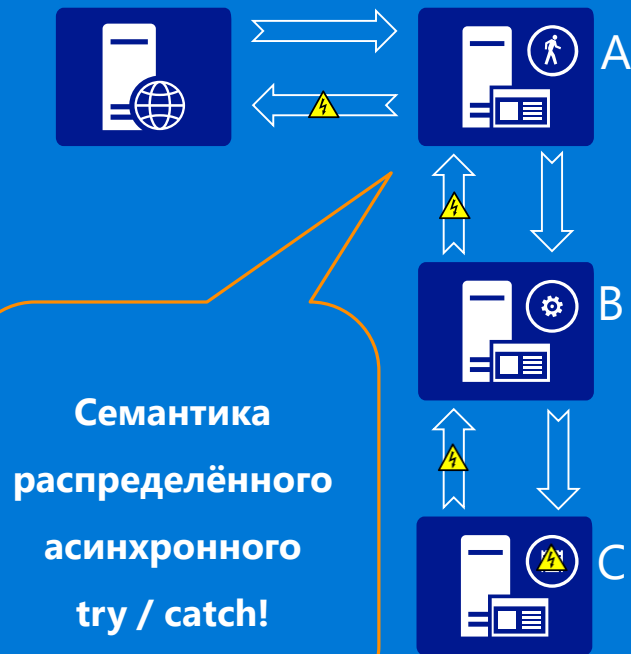
```
public interface IUser : IGrainWithGuidKey
{
    Task AddFriend(IUser friend);
}
```

```
IUser me = client.GetGrain<IUser>(myId);
IUser friend = client.GetGrain<IUser>(friendId);
```

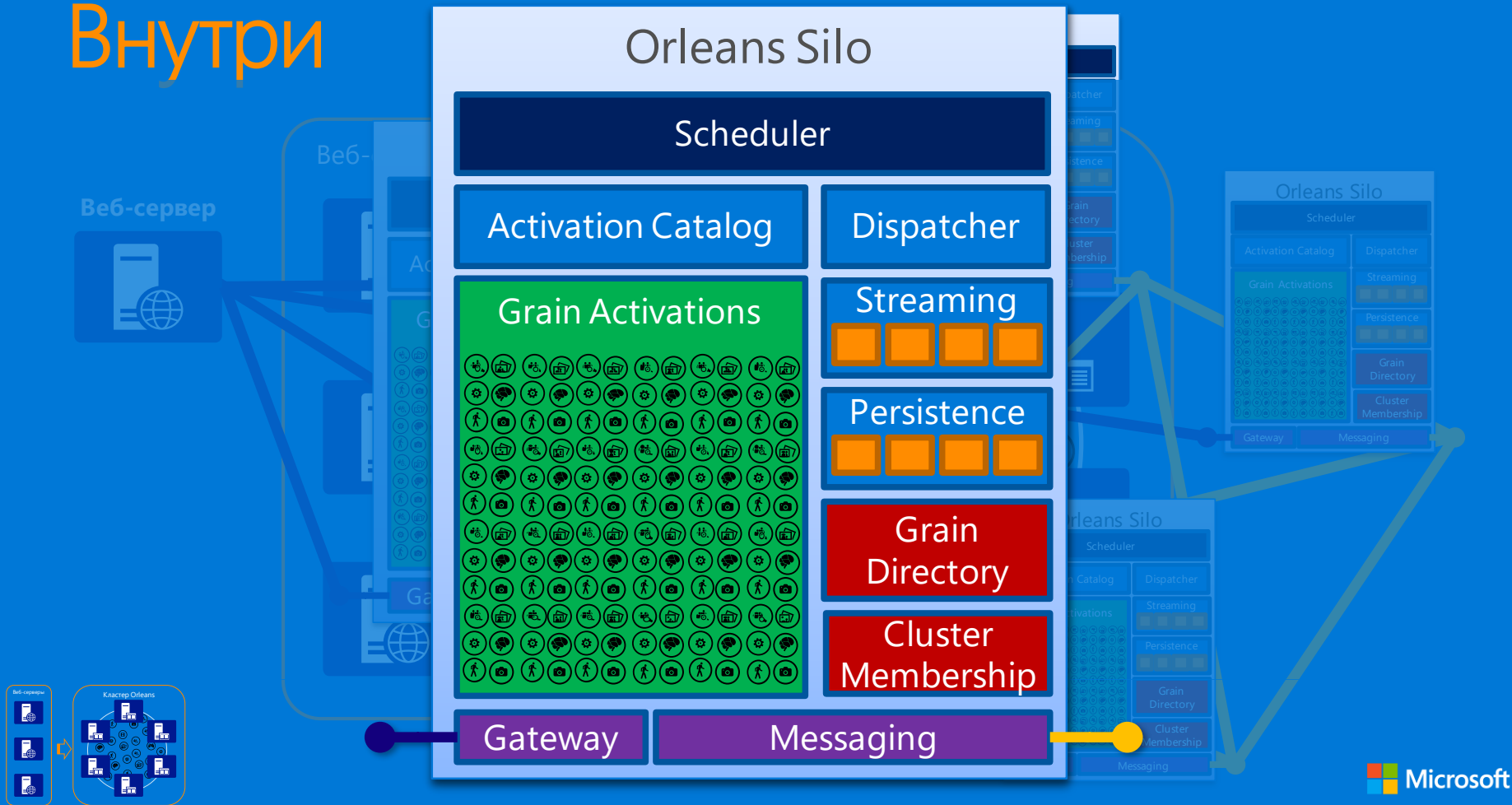
```
try
{
    await me.AddFriend(friend);
    Console.WriteLine($"Added friend {friendId}.");
}
catch (Exception e)
{
    Console.WriteLine(
        "Failed to add {friendId} as a friend: {e}.");
    throw;
}
```

Ссылка на грейн как аргумент

Обработка исключений



Внутри



ХОСТИНГ

Silo (Orleans сервер)

- Процесс или объект внутри процесса
- Два TCP порта: серверный и клиентский
- Конфигурация
- Код приложения (интерфейсы и классы грейнов)

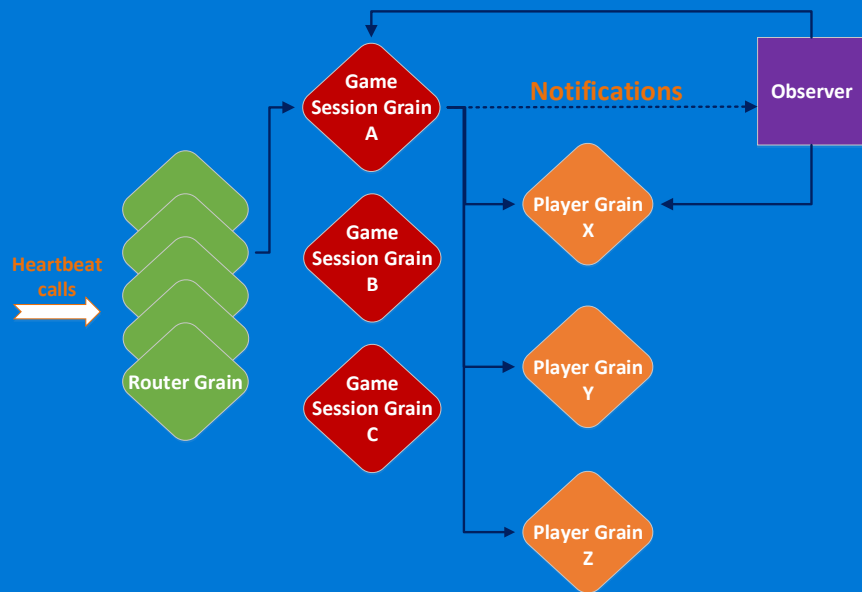
Встроенный протокол автоматического кластеринга

- Использует внешнее хранилище данных
- Взаимный мониторинг Silos и выявление отказов
- Автоматическое добавление в кластер новых Silos
- Клиенты (веб-сервера) подключаются ко всем Silos

Работает где угодно

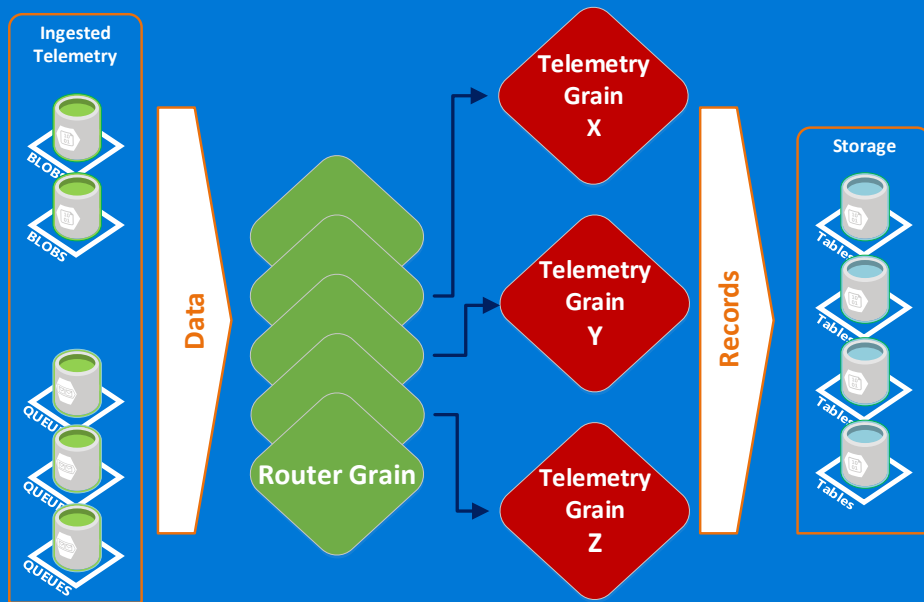
- Azure, Amazon AWS, private cloud, произвольный набор серверов
- Плагины кластеринга: Azure Table, SF, SQL (MS, My, Postgre, Oracle), ZooKeeper, Consul

Пример 1: Halo Presence



- *Router Grain* распаковывает и расшифровывает пакеты
- *Session Grain* аккумулирует информацию одного матча и оповещает *Player Grains* игроков
- Мобильный клиент через *Player Grain* находит *Session Grain* текущего матча и подписывается на оповещения о его ходе
- Миллионы параллельных матчей могут обрабатываться независимо
- Пакеты от разных Xbox-ов участников матча доставляются одному *Session Grain*

Пример 2: Обработка телеметрии Skype



- Телеметрия с клиентов записывается в Azure BLOBs+Queues
- *Router Grains* распаковывает блоки телеметрии и направляют *Telemetry Grains*
- Идентификатор *Telemetry Grain* является первичный ключ индивидуальной записи в Azure Table
- Благодаря 1:1 отношению между грейном и записью, избегаются конфликты по записи (eTag violations)
- Пропускная способность выросла в 4x
- До 750K сообщения в секунду

Рецепты успеха

Множество независимых контекстов

- Пользователи, сессии, устройства, каталоги
- Объект (грейн) для каждого контекста

Малая задержка, высокая пропускная способность

- Взаимодействие в реальном времени, в большом масштабе
- Состояние держится в памяти
- Малые затраты на IO

Горизонтальное взаимодействие

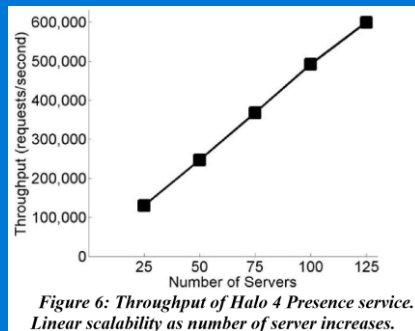
- Единое адресное пространство в пределах кластера
- Грейны могут напрямую вызывать друг друга и организовываться в графы

Автоматическое управление ресурсами

- Виртуальность грейнов – ключевой момент
- Автоматическая «сборка мусора» неактивных грейнов

Возврат к ООП

- Лучше соответствует реальному миру, чем stateless сервисы



Исходный код на GitHub

Этапы большого пути

Начат в Microsoft Research – 2009

В production – конец 2011

Выпуск Halo 4 – ноябрь 2012

Public preview – апрель 2014

GitHub (MIT) – январь 2015

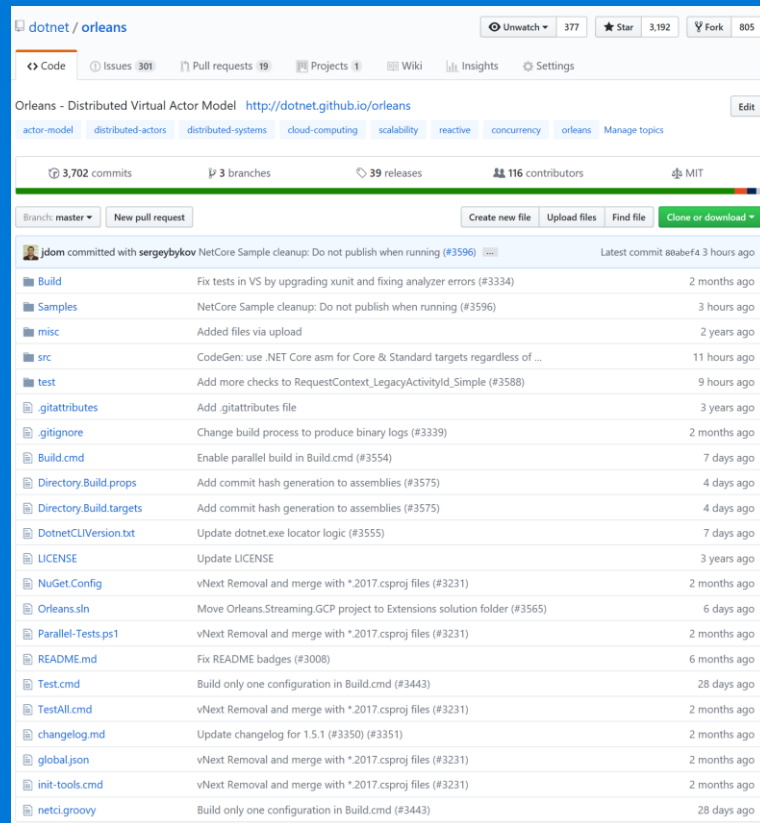
Открытая разработка на GitHub

Глобальное сообщество контрибьюторов

Один из самых популярных OSS проектов Microsoft

Кросс-платформенная v2.0.0 с CoreCLR

Продолжается сотрудничество с Research



dotnet / orleans

Unwatch 377 Star 3,192 Fork 805

Code Issues 301 Pull requests 19 Projects 1 Wiki Insights Settings

Orleans - Distributed Virtual Actor Model <http://dotnet.github.io/orleans>

actor-model distributed-actors distributed-systems cloud-computing scalability reactive concurrency orleans Manage topics

3,702 commits 3 branches 39 releases 116 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

jdom committed with sergeybykov	NetCore Sample cleanup: Do not publish when running (#3596)	Latest commit 80abef4 3 hours ago
Build	Fix tests in VS by upgrading xunit and fixing analyzer errors (#3334)	2 months ago
Samples	NetCore Sample cleanup: Do not publish when running (#3596)	3 hours ago
misc	Added files via upload	2 years ago
src	CodeGen: use .NET Core asm for Core & Standard targets regardless of ...	11 hours ago
test	Add more checks to RequestContext_LegacyActivityId_Simple (#3588)	9 hours ago
gitattributes	Add .gitattributes file	3 years ago
gitignore	Change build process to produce binary logs (#3339)	2 months ago
Build.cmd	Enable parallel build in Build.cmd (#3554)	7 days ago
Directory.Build.props	Add commit hash generation to assemblies (#3575)	4 days ago
Directory.Build.targets	Add commit hash generation to assemblies (#3575)	4 days ago
DotnetCLIVersion.txt	Update dotnet.exe locator logic (#3555)	7 days ago
LICENSE	Update LICENSE	3 years ago
NuGet.Config	vNext Removal and merge with *.2017.csproj files (#3231)	2 months ago
Orleans.sln	Move Orleans.Streaming.GCP project to Extensions solution folder (#3565)	6 days ago
Parallel-Tests.ps1	vNext Removal and merge with *.2017.csproj files (#3231)	2 months ago
README.md	Fix README badges (#3008)	6 months ago
Test.cmd	Build only one configuration in Build.cmd (#3443)	28 days ago
TestAll.cmd	vNext Removal and merge with *.2017.csproj files (#3231)	2 months ago
changelog.md	Update changelog for 1.5.1 (#3350) (#3351)	2 months ago
global.json	vNext Removal and merge with *.2017.csproj files (#3231)	2 months ago
init-tools.cmd	vNext Removal and merge with *.2017.csproj files (#3231)	2 months ago
netci.groovy	Build only one configuration in Build.cmd (#3443)	28 days ago

Имитация как высшая форма лести



What is Orbit?

Orbit is a framework to write distributed systems using virtual actors on the JVM. A virtual actor is an object that interacts with the world using asynchronous messages.

At any time an actor may be active or inactive. Usually the state of an inactive actor will reside in the database. When a message is sent to an inactive actor it will be activated somewhere in the pool of backend servers. During the activation process the actor's state is read from the database.

Actors are deactivated based on timeout and on server resource usage.

It is heavily inspired by the [Microsoft Orleans](#) project.

A screenshot of the Proto actor website. The header includes the "proto actor" logo and navigation links: HOME, TEAM, BLOG, DOCUMENTATION, CODE, and PROJECT CHAT. The main content area is titled "Virtual Actors" and describes a "Virtual Actor abstraction, which provides a straightforward approach to building distributed interactive applications, without the need to learn complex programming patterns for handling concurrency, fault tolerance, and resource management." To the right of the text is a diagram showing a hierarchical structure of actors: "Scale 0" at the top, followed by "Erlang Node", then "Host Partitions" (containing "Node 1" through "Node 5"), and finally "State Processes" (containing "Node 1" through "Node 5"). Arrows indicate the flow of communication between these components.

Introduction to Service Fabric Reliable Actors

06/29/2017 • 11 minutes to read • Contributors all

In this article

[What are Actors?](#)

[Actors in Service Fabric](#)

[Next steps](#)

Reliable Actors is a Service Fabric application framework based on the [Virtual Actor](#) pattern. The Reliable Actors API provides a single-threaded programming model built on the scalability and reliability guarantees provided by Service Fabric.

A screenshot of the "erleans" README file. The title "erleans" is followed by a green "PASSED" badge. Below the title is a section titled "Components". Further down is a section titled "Grains" which contains two paragraphs of text. The first paragraph describes "Stateful grains" and their activation process. The second paragraph describes "Grain state" persistence and the change id/etag mechanism.

Спасибо!

@SergeyBykov

@msftorleans

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

<https://gitter.im/dotnet/orleans>

