Microsoft

Reactor

Microsoft Reactor
Launch and approach

An Introduction to Orleans

David Gristwood & Ben Coleman
Cloud Solution Architects,
Microsoft

Agenda

Orleans

- An introduction to Orleans
- Smilr - a microservices showcase
- Coding Smilr with Orleans
- Deploying Smilr Orleans in AKS

.NET

An introduction to Orleans

The Actor model

- It is a programming model
  - In much the same vein as CQRS, Lambda, etc
  - Optimised for distributed, high performance, concurrent computation
  - Popularised by Erlang language (at Ericsson), Akka (Java, Scala and .NET)
- Reduce code complexity in complex systems
  - Deadlock and re-entrant issues handled by system
  - Asynchronous communications between actors
- Improve performance via state in middle tier
  - Middle layer isn’t just wrapper round database CRUD

Orleans

- Framework for distributed systems
  - Programming Model
  - Runtime
- From Microsoft Research, Open Source under .NET Foundation
- Built on .NET (Core), runs on:
  - Linux/Windows/macOS
  - Bare metal/Kubernetes/VMs/…
- Used by:
  - Microsoft (Azure, Xbox, Skype, others)
    - Halo, PlayFab, Gears of War, internal services
  - Honeywell, Gigya, Visa, NRK, TransUnion, Lebara, Drawboard, US NIST, others

Grains

- Fundamental building block
- Virtual Actor, always there
- Effectively: Distributed Object
- Unit of computation + state
- Send/receive messages
- React to incoming messages

Orleans: Distributed Virtual Actors for Programmability and Scalability

Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, Jorgen Thelin
Microsoft Research

Abstract

High-scale interactive services demand high throughput with low latency and high availability, difficult goals to meet with the traditional stateless 3-tier architecture. The actor model makes it natural to build a stateful middle tier and achieve the required performance. However, the popular actor model platforms still pass many distributed systems problems to the developers.

The Orleans programming model introduces the novel abstraction of virtual actors that solves a number of the complex distributed systems problems, such as reliability and distributed resource management, liberating the developers from dealing with those concerns. At the same time, the Orleans runtime enables applications to attain high performance, reliability and scalability.

This paper presents the design principles behind Orleans and demonstrates how Orleans achieves a simple programming model that meets these goals. We describe how Orleans simplified the development of several scalable production applications on Windows Azure, and report on the performance of those production systems.

required application-level semantics and consistency on a cache with fast response for interactive access.

The actor model offers an appealing solution to these challenges by relying on the *function shipping paradigm*. Actors allow building a stateful middle tier that has the performance benefits of a cache with data locality and the semantic and consistency benefits of encapsulated entities via application-specific operations. In addition, actors make it easy to implement horizontal, "social", relations between entities in the middle tier.

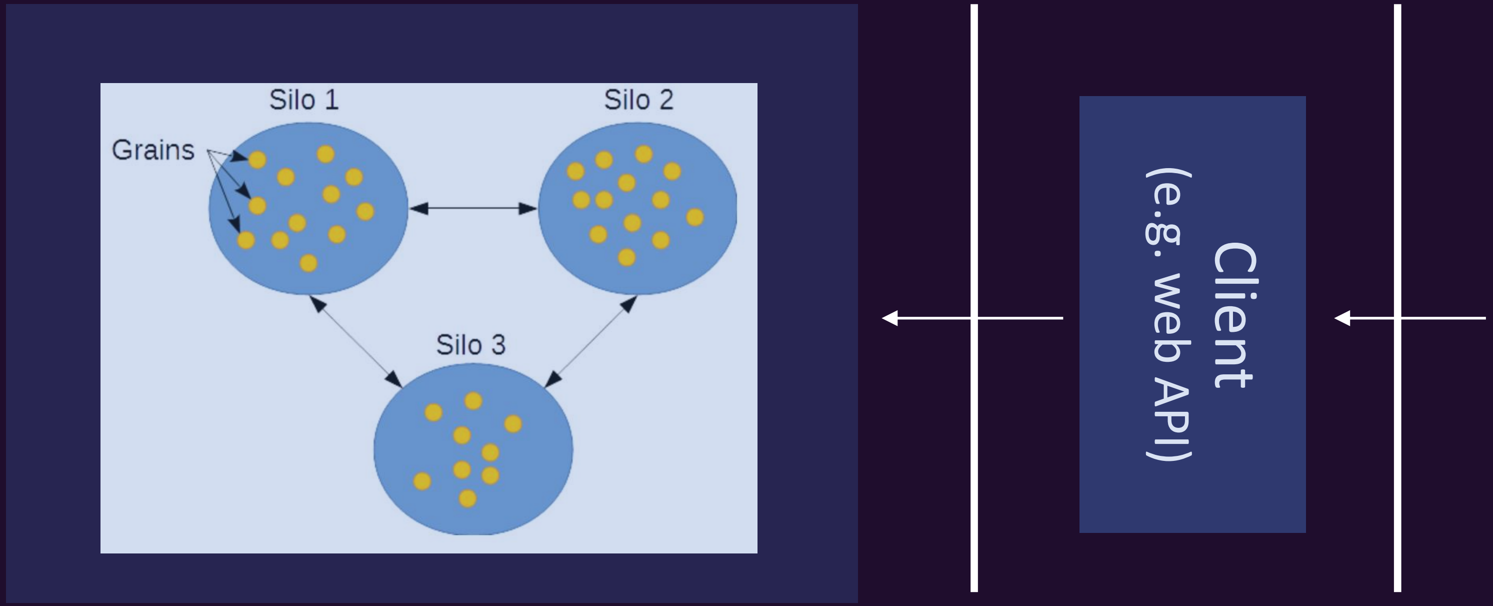
Another view of distributed systems programmability is through the lens of the object-oriented programming (OOP) paradigm. While OOP is an intuitive way to model complex systems, it has been marginalized by the popular service-oriented architecture (SOA). One can still benefit from OOP when implementing service components. However, at the system level, developers have to think in terms of loosely-coupled partitioned services, which often do not match the application's conceptual objects. This has contributed to the difficulty of building distributed systems by mainstream developers. The actor model brings OOP back to the system level with actors appearing to developers very much like the familiar model of interacting objects.

https://aka.ms/orleans-paper-2014

Grains

- Model objects in domain
  - IoT device twins, user profiles, game sessions, auction items, blog posts, …
- Create graphs for computation/communication
  - Fan-out, Fan-in & other processing patterns
  - Concurrent workers, caches
  - Distributed system realities
- Select an appropriate granularity

Grains live in Silos



- Orleans runtime does the heavy lifting
  - Manages silos and grain location
  - Transparently instantiates and garbage collects actors
  - Routes requests & responses and invokes methods on grain
  - Transparently recovers from server failures

in-memory,
persisted, or
stateless

`User/davidgri`

grain = identity + behavior [+ state]

`class User : Grain, IUser`

Interface

```
public interface IUser : IGrainWithStringKey
{
  Task SendMessage(IUser sender, string body);

  Task<List<Message>> GetMessages();
}
```

Grain

```csharp
public class User : Grain, IUser
{
  IGrainState<List<Message>> _messages; // hold messages inside grain

  public User([GrainState("messages")] IGrainState<List<Message>> messageState)
    => _messages = messageState;

  async Task SendMessage(IUser sender, string body)
  {
    _messages.Value.Add(new Message { Sender = sender, Body = body });
    await _messages.WriteStateAsync(); // persist state
  }

  Task<List<Message>> GetMessages() => Task.FromResult(_messages.Value);
}
```

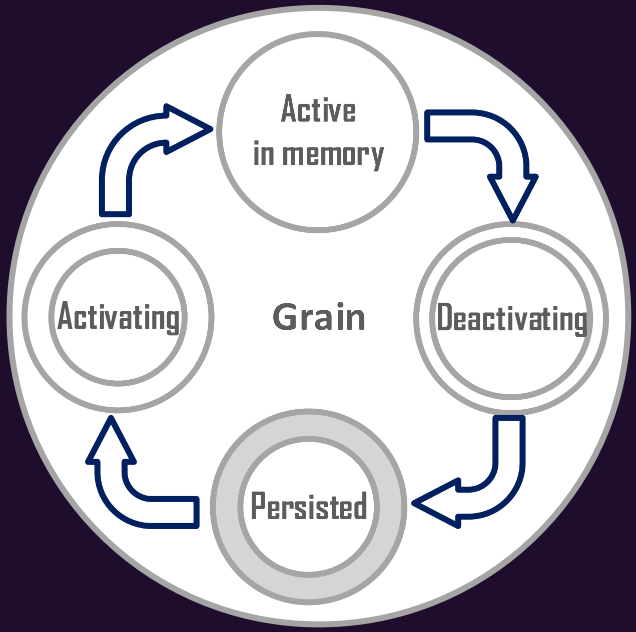
Methods are always single-threaded

State is in grain

Client

```
// Get grain references
IUser me = client.GetGrain<IUser>(“davidgri");
IUser friend = client.GetGrain<IUser>(“reuben");

// Call a grain to send message
await friend.SendMessage(sender: me, body: “Hello Orleans");

// Call a grain to retrieve all messages
var messages = await me.GetMessages();
foreach (var msg in messages)
{
  Console.WriteLine($"{msg.Sender.GetPrimaryKeyString()} says: {msg.Body}");
}
```

Grain lifecycle



Declarative persistence model via plugins for different storage services:

- ADO.NET for SQL, etc
- Azure Table, Blob, CosmosDB
- DynamoDB, and more…

Define internal state class and derive grain from it:

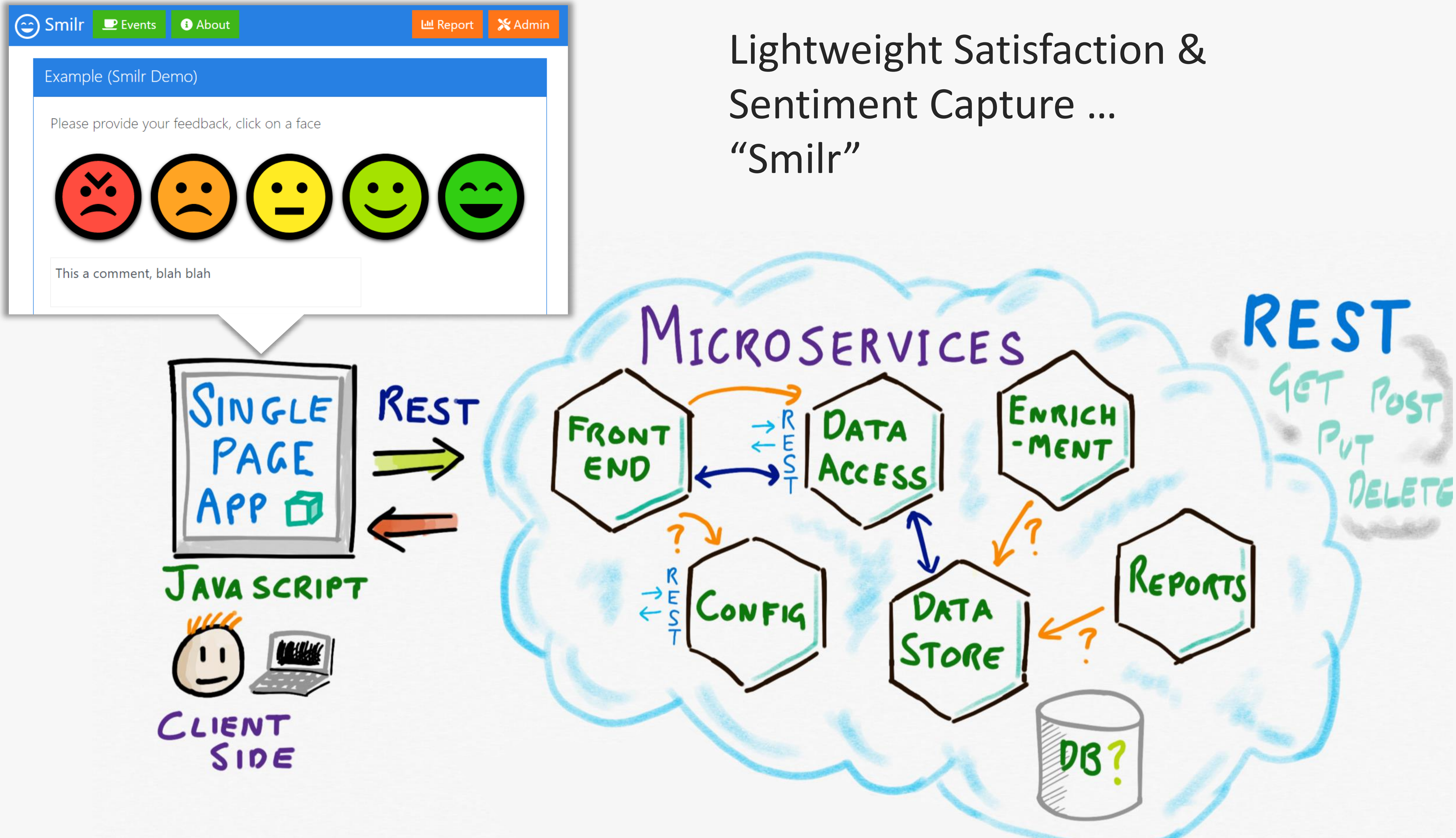
```
[StorageProvider(ProviderName="store1")]
public class MyGrain : Grain<MyGrainState>, /*...*/
{
  /*...*/
}
```

Smilr - a microservices showcase

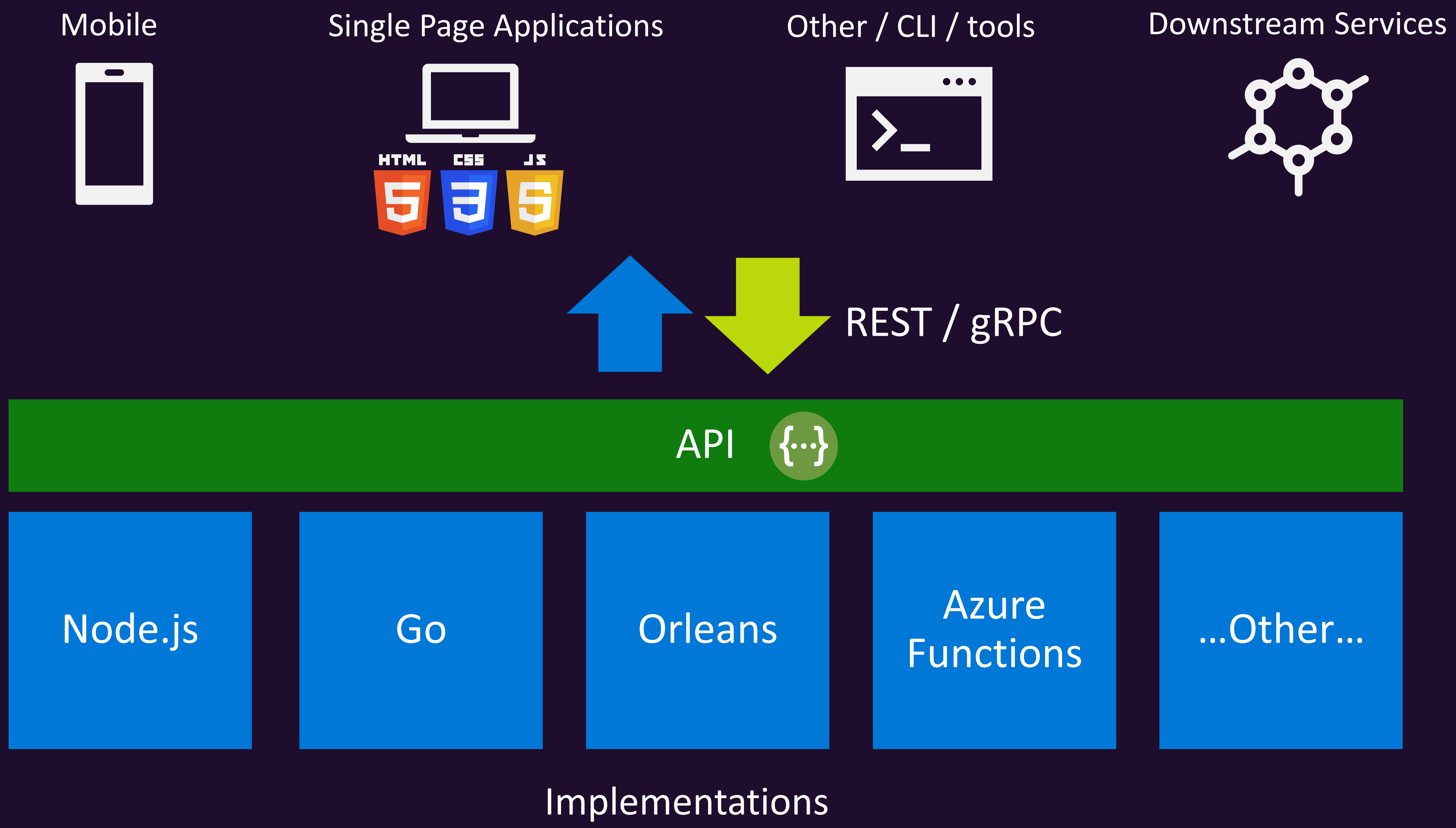
Project Goals

- Showcase for Azure compute scenarios
- Tie together disparate architecture choices
- Develop a real working system
- Explore use of new design patterns and technologies

Lightweight Satisfaction & Sentiment Capture … “Smilr”



Modern Application Architecture

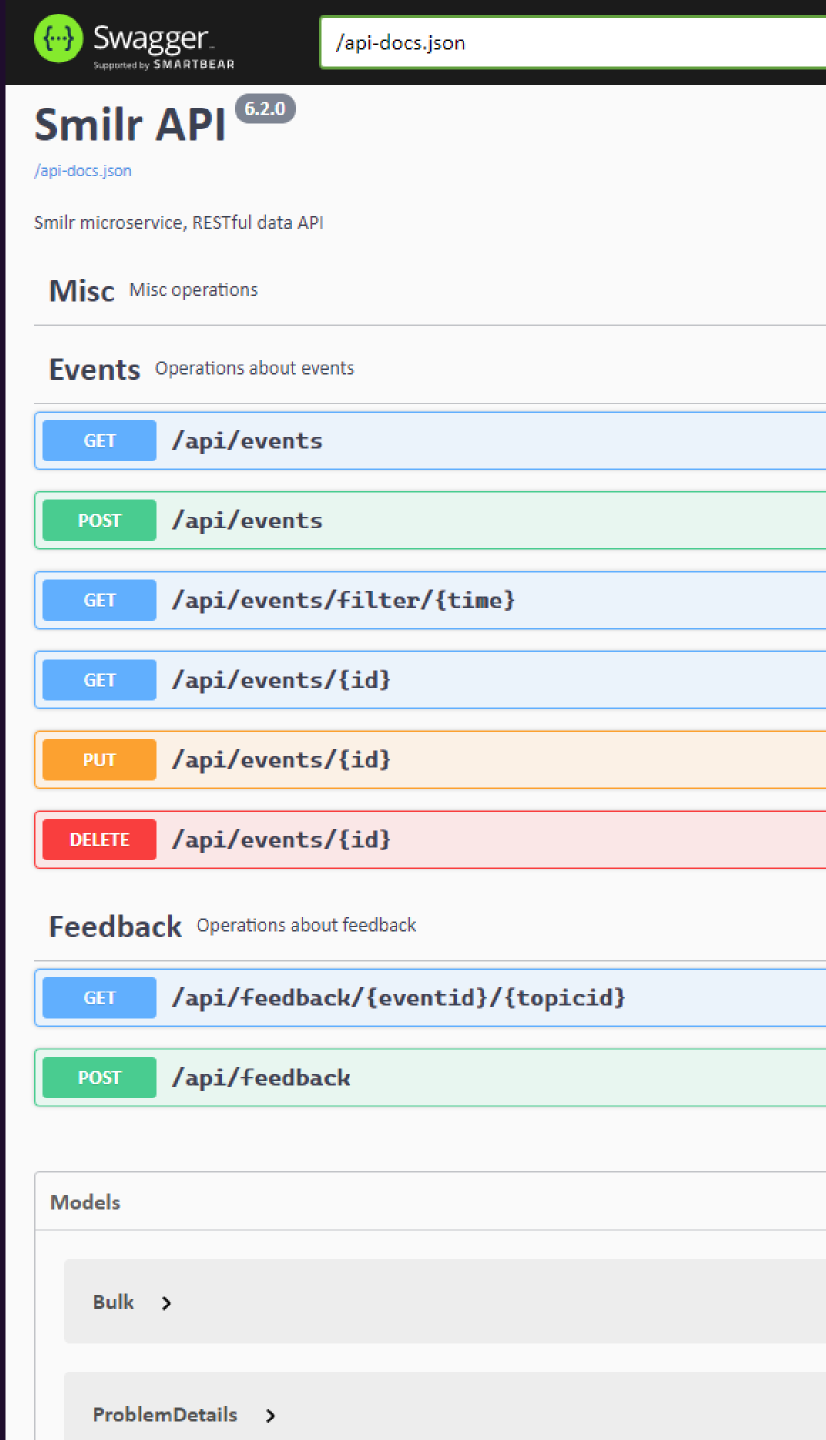


API as a Contract

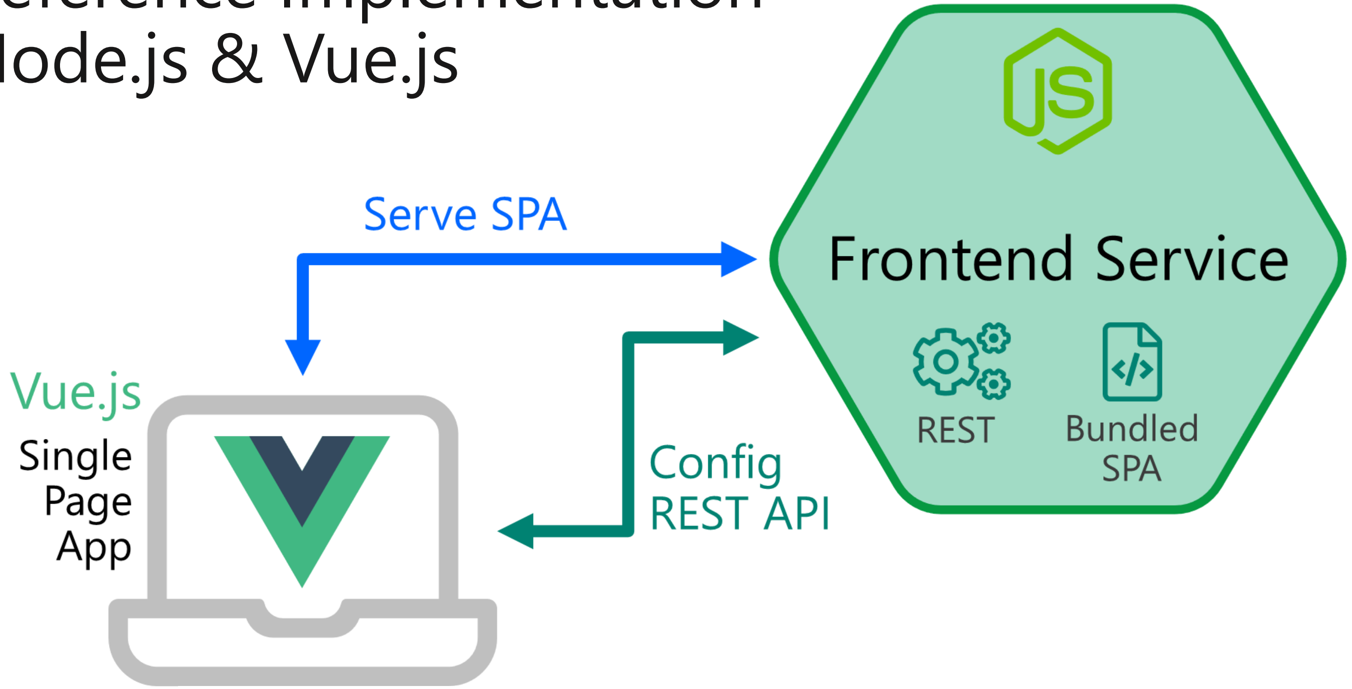
- Agree the shape of your API
  - Build implementations
  - Build clients
- Services, operations & models (entities)
- Swagger / OpenAPI for REST
  - Use auto generation & tools where possible
- Proto3 (.proto files) for gRPC

https://docs.microsoft.com/azure/architecture/best-practices/api-design

https://docs.microsoft.com/azure/architecture/best-practices/api-implementation

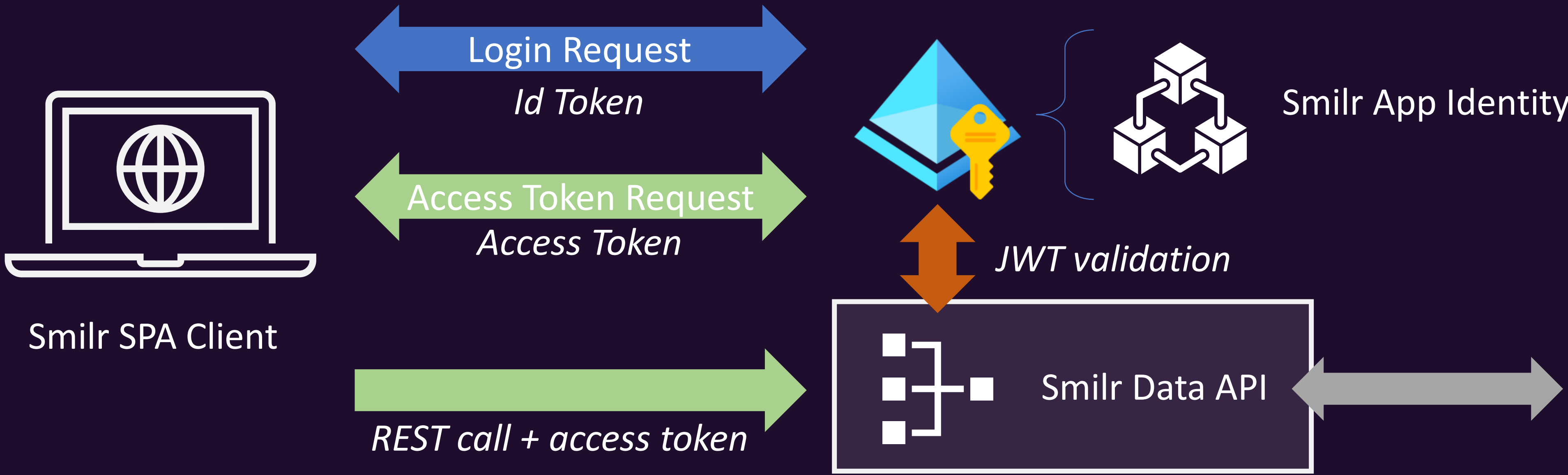


Reference Implementation
Node.js & Vue.js



Securing API Driven Apps

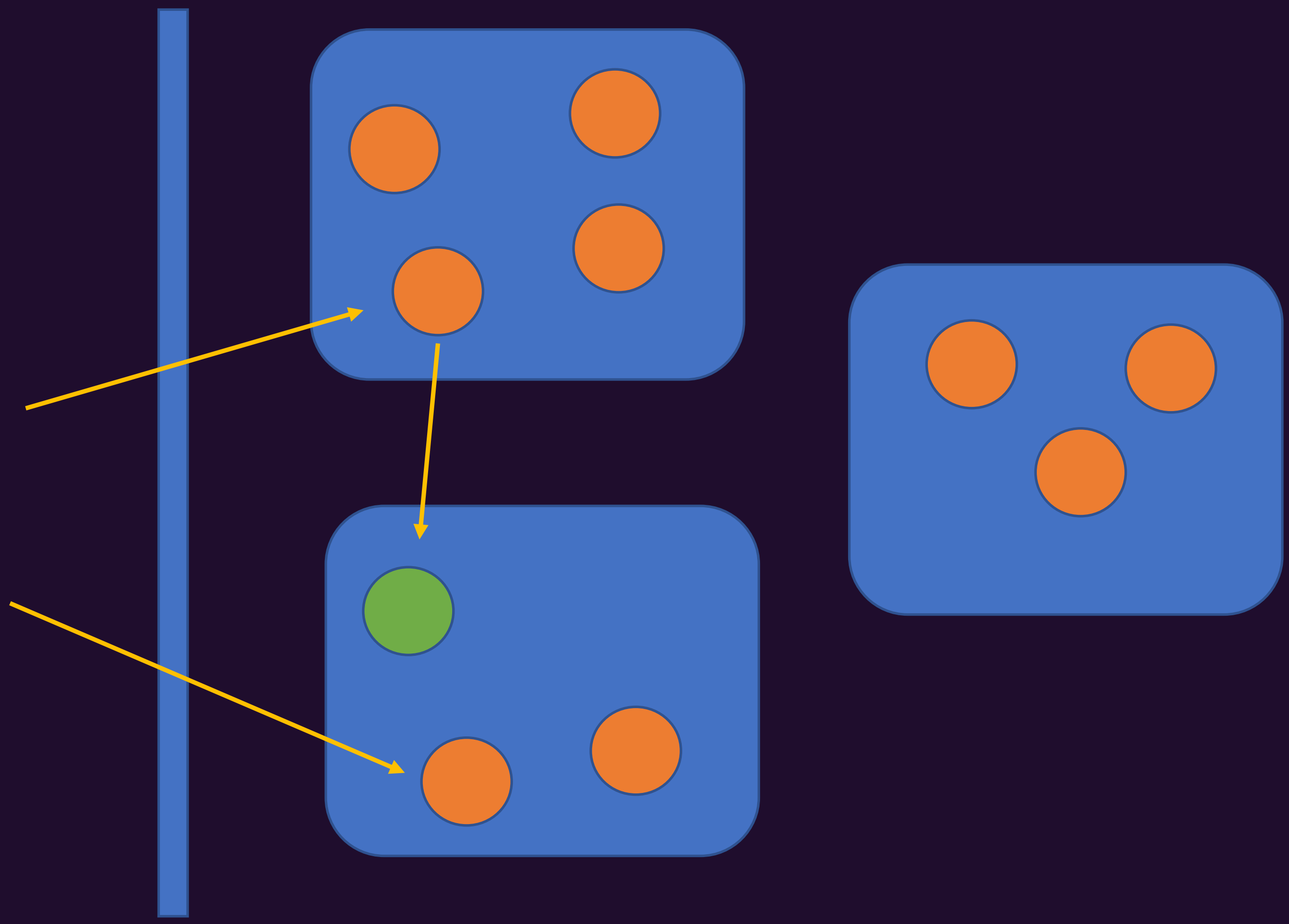
- Smilr API security journey
  - None – Bad!
  - One time passwords (OTP) & static keys – Still bad (but easy)!
  - OAuth 2.0 Bearer Tokens & JWT – Good!
- Securing SPAs is a challenge – OIDC/OAuth: Implicit Flow & PKCE



Coding Smilr with Orleans

Orleans and Smilr

- Replace backend database with Orleans grains
- C#, .NET Core 3.1, Orleans 3.2
- MVC Web API implements Smilr API and is the silo client
- Two grains types:
  - One for each event
  - One per system for aggregation
- More on the aggregator grain later...



MVC Web API

```
// POST /api/events - Create new event
[HttpPost("")]
0 references
public async Task<EventApiData> Post([FromBody] EventApiData body)
{
    // create new event code, which we tend to keep short to be more memorable
    string eventCode = makeId(body.title, 6);
    logger.LogInformation($"-- POST /api/events: Create new event, incoming body title = {body.title}

    // initialise grain with event info
    var grain = this.client.GetGrain<IEventGrain>(eventCode);
    await grain.Update(body.title, body.type, body.start, body.end, body.topics);

    // return body with event code added
    body._id = eventCode;
    return body;
}
```

```
// externally facing web API Event model
21 references
public class EventApiData
{
    4 references
    public string _id { get; set; }          // event key used to reference the event
    7 references
    public string title { get; set; }        // event title
    4 references
    public string type { get; set; }         // event type ('event', 'workshop', ...)
    4 references
    public string start { get; set; }        // ISO 8601 - YYYY-MM-DD
    4 references
    public string end { get; set; }          // ISO 8601 - YYYY-MM-DD
    3 references
    public TopicApiData[] topics { get; set; }  // list of topics, must be at least one
}
```

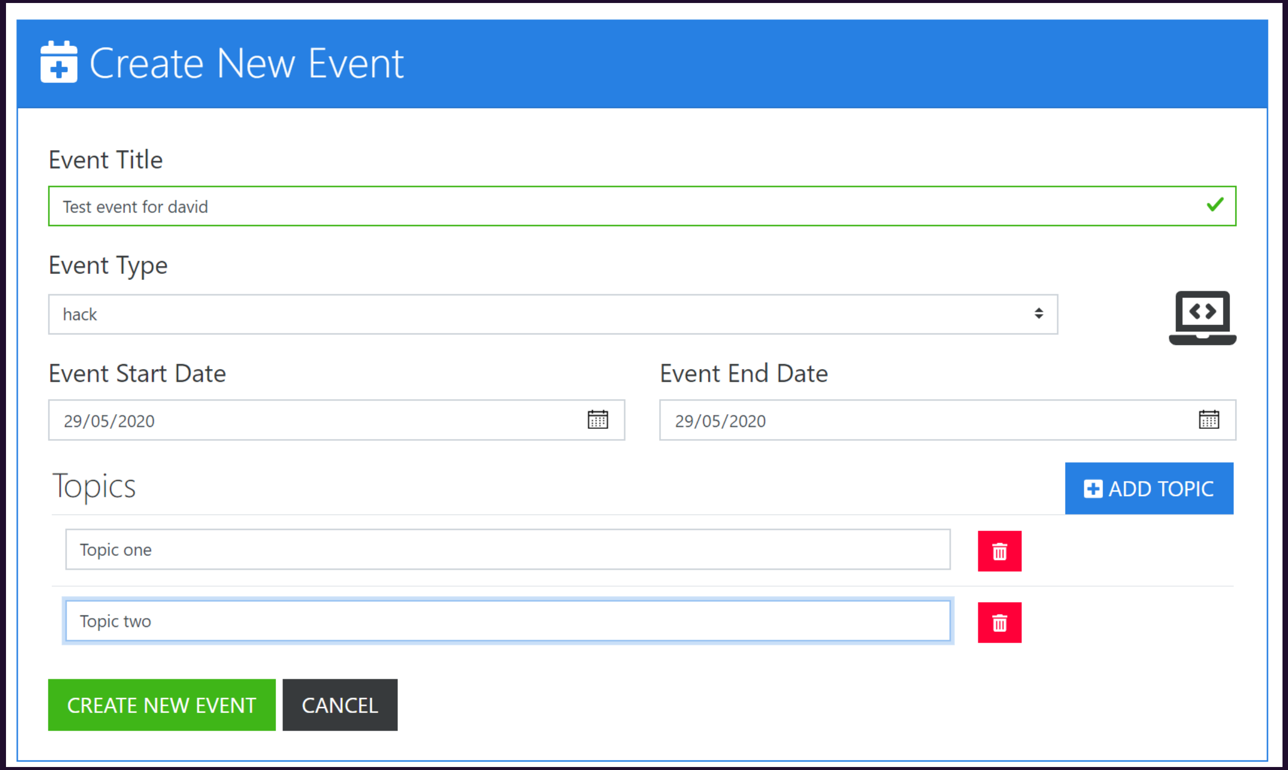
Smilr API

```
// POST /api/events - Create new event
[HttpPost("")]
0 references
public async Task<EventApiData> Post([FromBody] EventApiData body) ···

// PUT /api/events/{eventCode} - Update an existing event
[HttpPut("{eventCode}")]
0 references
public async Task<EventApiData> Put([FromBody] EventApiData body, string eventCode) ···

// GET /api/events/{id} - Get specific event info
[HttpGet("{id}")]
0 references
public async Task<EventApiData> GetEvent(string id) ···

// GET /api/events - return a list of all events
[HttpGet("")]
0 references
public async Task<EventApiData[]> GetEvents() ···

// GET /api/events/filter/{active|future|past} - return a list of events filtered to timeframe
[HttpGet("filter/{filter}")]
0 references
public async Task<EventApiData[]> GetFilteredEvents(string filter) ···
```

```
// GET /api/feedback/{eventid}/{topicid} - Return all feedback for specific event and topic
[HttpGet("{eventid}/{topicid}", Name = "Get")]
0 references
public async Task<FeedbackApiData[]> Get(string eventid, int topicid) ···

// POST /api/Feedback - submit feedback for an event + topic
[HttpPost]
0 references
public async Task Post([FromBody] FeedbackApiData body) ···
```

Event Grain

- One grain per event
- Event id is grain key
- Internal (persisted) state hold all event and feedback information



Event Grain

```
[StorageProvider(ProviderName = "grain-store")]
1 reference
public class EventGrain : Grain<EventGrainState>, IEventGrain
{
```

```
7 references
public interface IEventGrain : IGrainWithStringKey
{
    // initialise/update a grain with the core event info
    2 references
    Task Update(string title, string type, string start, string end, TopicApiData[] topics);

    // return event info
    1 reference
    Task<EventApiData> Info();

    // submit event + topic feedback
    1 reference
    Task<int> SubmitFeedback(int topic, int rating, string comment);

    // get all feedback for specific topic
    1 reference
    Task<FeedbackApiData[]> GetFeedback(int topicid);
}
```

These are the methods that the Event grain supports

```
// persisted event info
1 reference
public class EventGrainState
{
    // core event data
    0 references
    public string id { get; set; }
    2 references
    public string title { get; set; }
    2 references
    public string type { get; set; }
    2 references
    public string start { get; set; }
    2 references
    public string end { get; set; }
    4 references
    public TopicApiData[] topics { get; set; }

    // user feedback, added incrementally
    7 references
    public List<FeedbackGrainState> feedback { get; set; }
}
```

This is the internal state for the event grain that get persisted

Event Grain

```
public async Task Update(string title, string type, string start, string end, TopicApiData[] topics)
{
    string id = this.GetPrimaryKeyString();  // remember, the grain key is the event id

    // update interal grain state

    State.title = title;
    State.type = type;
    State.start = start;
    State.end = end;
    State.topics = topics;
    State.feedback = new List<FeedbackGrainState>();  //  lets clear all feedback, just to keep things simple
    await base.WriteStateAsync();  // persist internal state

    // update aggregator grain with info about this new event

    SummaryEventInfo eventInfo = new SummaryEventInfo();
    eventInfo.id = id;
    eventInfo.title = title;
    eventInfo.start = start;
    eventInfo.end = end;

    IAggregatorGrain aggregator = GrainFactory.GetGrain<IAggregatorGrain>(Guid.Empty);  // the aggregator grain is a singleton
    await aggregator.AddAnEvent(eventInfo);

    return;
}
```

Indexing and queries in Orleans

- Cross grain querying is expensive and not recommended
- Solution requires a CQRS style approach – separate reads from writes
- Two main options:
  - ‘Exhaust’ de-normalised, query optimised data to external store, such as SQL, Cosmos DB, etc
  - Aggregator grain that knows about all grains that can be queried
- Aggregator chosen to show off inter-grain communication
  - Real world implementation would have one aggregator per silo, fronted by one cross silo aggregator

Aggregator Grain

- Maintains list of all events
- Holds the minimum query information
- Query logic has to be coded

```
public interface IAggregatorGrain : IGrainWithGuidKey
{
    // does a specfic event exist? -1 no, >=0 yes
     1 reference
     Task<int> IsAnEvent(string id);

    // add a new event to the list of events
    1 reference
    Task AddAnEvent(SummaryEventInfo eventInfo);

    // delete an event from the list of events
    0 references
    Task DeleteAnEvent(string id);

    // return array of events matching passed in filter
    2 references
    Task<EventApiData[]> ListEvents(string filter);
}
```

```
// persisted aggregator summary event info list
1 reference
public class AggregatorGrainState
{
    15 references
    public List<SummaryEventInfo> allevents { get; set; }  // list of  events and key info needed for filtering
}
```

Aggregator Grain

```
public async Task AddAnEvent(SummaryEventInfo eventInfo)
{
    logger.LogInformation($"** AggregatorGrain:AddAnEvent() for event id {eventInfo.id}, title {eventInfo.title}");

    // ensure allevent List is constructed ok
    if (State.allevents == null)
        State.allevents = new List<SummaryEventInfo>();

    // check to see if event already exists, and if so, removed, before inserting new event info
    int index = State.allevents.FindIndex(item => item.id == eventInfo.id);
    if (index >= 0)
        State.allevents.RemoveAt(index);

    // add this event key to our active list
    State.allevents.Add(eventInfo);
    logger.LogInformation($"** AggregatorGrain:AddAnEvent() about to write WriteStateAsync for new event id {eventInfo.id}");
    await base.WriteStateAsync();

    return;
}
```

Deploying Smilr Orleans in Kubernetes

Smilr in Kubernetes

Example non-Orleans Deployment Architecture

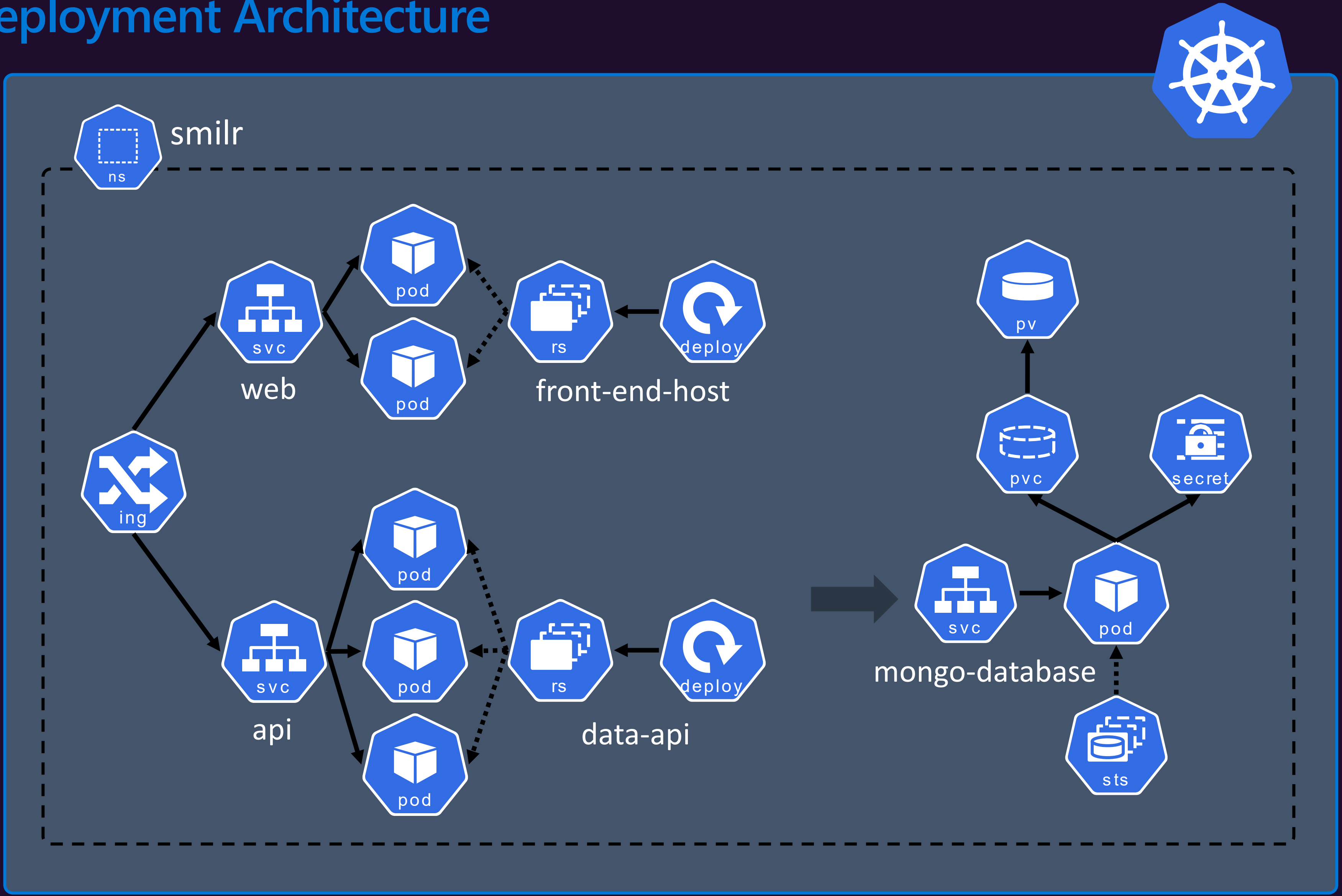
Microservices pattern is an ideal candidate for Kubernetes

Stateless components (API & frontend) running as replicated pods

Ingress routing traffic to web and API pods

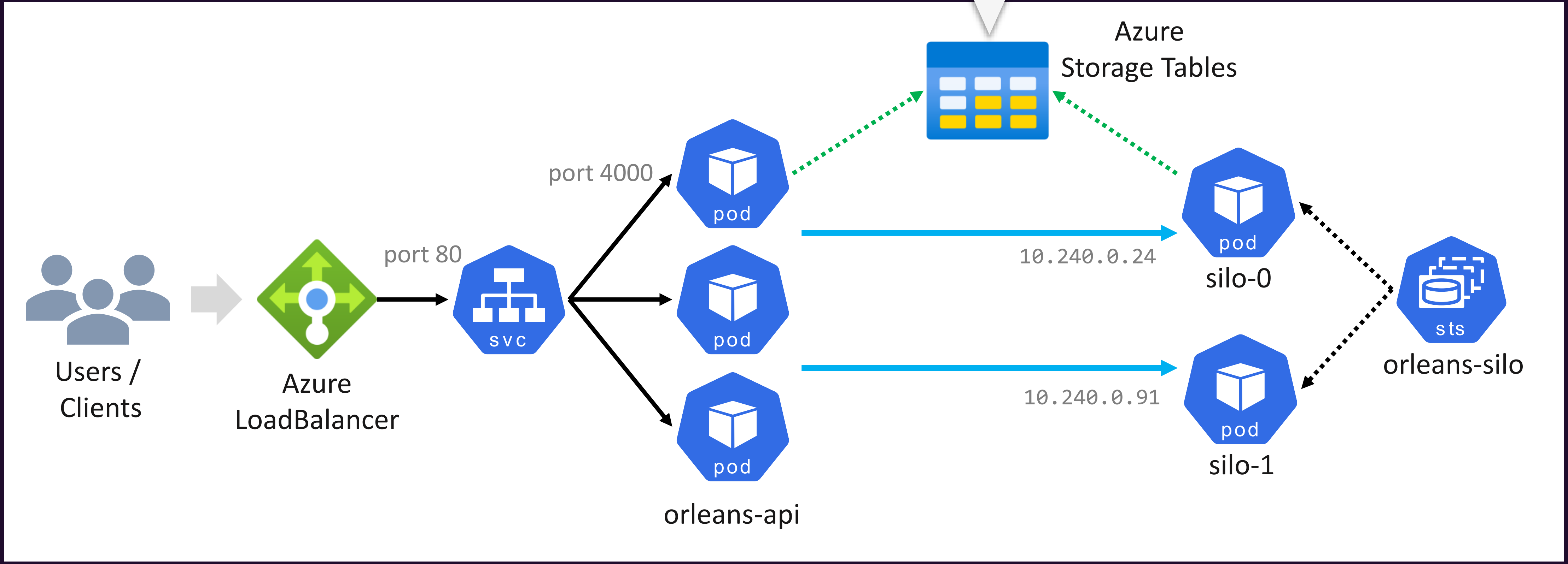
Only the Ingress is exposed to the internet

Database running in StatefulSet, with internal only service and persistence



Orleans in Kubernetes

PartitionKey	Address	Status	DeploymentId	FaultZone	HostName	IAmAliveTime	InstanceName	Port	ProxyPort	RoleName	SiloName
smilr-kube	10.240.0.24	Active	smilr-kube	0	orleans-silo-0	2020-07-23 08:20:34.990 GMT	Silo_3b01d	11111	30000	Silo	Silo_3b01d
smilr-kube	10.240.0.91	Active	smilr-kube	0	orleans-silo-1	2020-07-23 08:21:56.101 GMT	Silo_da0ac	11111	30000	Silo	Silo_da0ac
smilr-kube	*null*	*null*	smilr-kube	*null*	*null*	*null*	*null*	*null*	*null*	*null*	*null*



Silo Deployment

- Uses StatefulSets
- Runs a HA cluster (two pods)
- Has no service in front of it
- Exposes Orleans ports 30000 & 11111
- Connects to Azure Storage for cluster synchronization
- Kubernetes secret holds storage connection string

```yaml
kind: StatefulSet
apiVersion: apps/v1
metadata:
  name: orleans-silo
  labels:
    app: orleans-silo
spec:
  replicas: 2
  serviceName: orleans-silo
  selector:
    matchLabels:
      app: orleans-silo
  template:
    metadata:
      labels:
        app: orleans-silo
    spec:
      containers:
      - name: orleans-silo
        image: demo.azurecr.io/smilr-orleans/silo:latest
        imagePullPolicy: Always

        ports:
        - name: gateway
          containerPort: 30000
        - name: silo
          containerPort: 11111

        env:
        - name: Orleans__ClusterId
          value: smilr-kube
        - name: Orleans__ConnectionString
          valueFrom:
            secretKeyRef:
              name: orleans-secrets
              key: storageConnectionString
```

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: orleans-api
  labels:
    app: orleans-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: orleans-api
  template:
    metadata:
      labels:
        app: orleans-api
    spec:
      containers:
      - name: orleans-api
        image: demo.azurecr.io/smilr-orleans/api:latest
        imagePullPolicy: Always

        ports:
        - containerPort: 4000

        env:
        - name: Orleans__ClusterId
          value: smilr-kube
        - name: Orleans__ConnectionString
          valueFrom:
            secretKeyRef:
              name: orleans-secrets
              key: storageConnectionString
```

```yaml
kind: Service
apiVersion: v1
metadata:
  name: data-api-svc
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 4000
  selector:
    app: orleans-api
```

API Deployment

- Uses Deployments (it’s stateless)
- Runs in 3 pods replicated
- Has LoadBalancer service in front of it – external
- Exposes ASP.Core Kestrel port 4000
- Connects to Azure Storage for silo and cluster access


KUBERNETES
CLUSTERS
bcdemo
Namespaces
Nodes
Workloads
Deployments
orleans-api
StatefulSets
orleans-silo
DaemonSets
Jobs
CronJobs
Pods
orleans-api-64d8c4b8c9-...
orleans-silo-0
orleans-silo-1
Network
Services
data-api-svc
kubernetes
silo-gateway-svc
Endpoints
Ingress
Storage
Configuration
Custom Resources
Helm Releases
bcdemo-admin
HELM REPOS
CLOUDS
rest client test scripts.http
orleans > rest client test scripts.http
get a list of all events
Send Request
GET http://{{host}}/api/events HTTP/1.1
create event (in the past)
Send Request
POST http://{{host}}/api/events HTTP/1.1
content-type: application/json
{
"title": "demo event",
"type": "event",
"start": "2020-01-01",
"end": "2020-01-01",
"topics": [
{
"id": 1,
Response(1266ms)
HTTP/1.1 200 OK
Connection: close
Date: Fri, 31 Jul 2020 08:10:58 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
DEBUG CONSOLE
OUTPUT
TERMINAL
❯ k logs orleans-silo-0 -f
Starting Silo...
Using ClusterId=smilr-kube, ServiceId=smilr, siloPort=11
warn: Orleans.Runtime.Silo[100405]
Note: Silo not running with ServerGC turned on - recomme
time>-<gcServer enabled="true">
warn: Orleans.Runtime.Silo[100405]
Note: ServerGC only kicks in on multi-core systems (sett
le-core machines).
warn: Orleans.Runtime.NoOpHostEnvironmentStatistics[100708]
No implementation of IHostEnvironmentStatistics was foun
** EventGrain Update()for event id = 8lHoqk, with title demo e
** EventGrain Update() about to write WriteStateAsync
orleans-api
orleans-secrets
orleans-silo
orleans-api-64d8c4b8c9
0
1
x87lf
data-api-svc
silo-gateway-svc
51.104.175.206

Orleans Resources

- Get started at https://dotnet.github.io/orleans/
- Try the samples at https://github.com/dotnet/orleans/tree/master/Samples
- Read the docs at https://dotnet.github.io/orleans/Documentation/
- Contribute at https://github.com/dotnet/orleans

Slides for this session https://github.com/benc-uk/smilr

Microsoft

Reactor

LONDON

Thank you for attending today's session.
We would like to ask that you ***please*** complete a short survey:

https://aka.ms/Reactor/Survey

Event Code: XXXXX

meetup.com/Microsoft-Reactor-London