Orleans Best Practices

Agenda

- Scenarios & General Fit
- Designing Grains
- Implementing Grains
- Persistence
- Deployment & Production Management
- Logging & Testing
- Troubleshooting

Scenarios & General Fit

- Consider Orleans when you have
  - Significant number of loosely coupled entities (hundreds to millions)
  - Entities are small enough to be single-threaded
  - Workload is interactive: request-response, start/monitor/complete
  - Need or may need to run on >1 server
  - No need for global coordination, only between a few entities at a time
  - *Different entities used at different times
- Problematic fit
  - Entities need direct access to each other's memory
  - Small number of huge entities, multithreaded
  - Global coordination/consistency needed
  - *Long running operations, batch jobs, SIMD

* it depends

Designing Grains

- Actors are not object, although very similar
- Loosely coupled, isolated, mostly independent
  - Encapsulate and manage their state independently from other grains
  - Can fail independently
- Avoid chatty interfaces between grains
  - Message passing is much more expensive than direct memory access
  - If two grains constantly talk to each other, maybe they should be one
  - Consider size and complexity of arguments, serialization
    - Sometimes it’s cheaper to resend a binary message and deserializes it twice
- Avoid bottleneck grains
  - Single coordinator/registry/monitor
  - Do staged aggregation if necessary

Implementing Grains -- Asynchrony

- Everything has to be async (TPL), no thread-blocking operations
- *await* is the best mechanism to compose async operations
- Typical cases:
  - Return a concrete value:
    ```
    return Task.FromResult(value);
    ```
  - Return a Task of the same type:
    ```
    return foo.Bar();
    ```
  - Await a Task and continue execution:
    ```
    var x = await bar.Foo();
    var y = DoSomething(x);
    return y;
    ```
  - Fan-out:
    ```
    var tasks = new List<Task>();
    foreach(var grain in grains)
       tasks.Add(grain.Foo());
    await Task.WhenAll(tasks);
    DoMore();
    ```

Implementing Grains

- When to use [StatelessWorker]
  - Functional operations: decrypt, decompress, before forwarding for processing
  - Multiple activations, always local
  - E.g., good for staged aggregation (locally within silo first)
- By default grains are non-reentrant
  - Deadlock in case of call cycles, e.g. call itself
  - Deadlocks are automatically broken with timeouts
  - [Reentrant] to make a grain class reentrant
  - Reentrant is still single-threaded but may interleave
  - Dealing with interleaving is error prone
- Inheritance
  - Inheritance of grain interfaces is easy
  - Multiple grain classes implementing same interface may require disambiguation
  - Limited inheritance of grain classes
    - Declarative persistence breaks inheritance
- Generics are supported

Grain Persistence Overview

Orleans grain state persistence APIs are designed to provide extensible storage functionality with easy-to-use API.

- Tutorial: https://orleans.codeplex.com/wikipage?title=Declarative%20Persistence&referringTitle=Step-by-step%20Tutorials

Overview – Grain State Persistence

- Define .NET interface extending `Orleans.IGrainState` containing fields to be included in grain's persisted state.
- Grain class should extend `GrainBase<T>` and adds strongly typed `State` property to the grain's base class.
- The first `State.ReadStateAsync()` will occur automatically before `ActivateAsync()` is called for a grain.
- Grain should call `State.WriteStateAsync()` whenever they change data in the grain's state object
  - Grains typically call `State.WriteStateAsync()` at the very end of grain method, and return the Write promise.
  - Storage provider _*could*_ try to **batch** Write's for efficiency, but behavioral contract & config is orthogonal to storage API used by grain.
  - Alternatively grains might use **timer** to only write updates periodically.
    Application can decide how much "eventual consistency" / staleness it can allows – range from immediate / none to several minutes.
- Each grain class can only be associated with **one** storage provider.
  - The particular provider to use for a grain defined with `[StorageProvider(ProviderName="name")]` attribute.
  - Silo config file needs `<StorageProvider>` entry in silo config file with corresponding **name** -- see tutorial above for example.
  - Storage provider may be composite provider, Example: ShardedStorageProvider

Storage Providers

Built-in Storage Providers

- All built-in storage providers live in the **`Orleans.Storage`** namespace from **OrleansProviders.dll**.
- **`MemoryStorage`** is ONLY for debug / unit testing – Data stored in-memory with no durable persistence
- **`AzureTableStorage`** stores data in Azure table storage
  - Configure with Azure storage account info + optional DeleteStateOnClear [hard vs soft delete]
  - Data stored in binary format in one Azure table cell using efficient Orleans serializer.
    Data size limit == max size of Azure table column == 64KB binary data.
    Community contributed code extends to use multiple table columns, for overall max 1MB.
- **`ShardedStorageProvider`** writes data across a number of underlying storage providers, based on grain id hash.
  - Usage example: https://orleans.codeplex.com/discussions/546730

Storage Provider Debug Tips

- Turn on **`TraceOverride Verbose3`** logging in silo config file for built-in storage providers to get much more info about what is happening with storage operations.
  - `LogPrefix="Storage"` for all providers, or specific type using "`Storage.Memory`" / "`Storage.Azure`" / "`Storage.Shard`".
- Can use **Fiddler** to debug & optimize REST API calls to/from Azure storage. http://t.co/JV8N7fgW5k

Dealing With Failure of Storage Operations

- Either grains or storage providers can await storage operations and **retry any** failures if desired.
- If unhandled, failure will be propagated back to **caller** / client as a broken promise.
- No concept currently of activations getting destroyed automatically if storage operation fails [except initial Read]
- Built-in storage providers **do not** retry failing storage operations by default.

Grain Persistence – Hints & Tips

Grain Sizing

- For throughput, usually better to use **many smaller grains** than few large grains,
  but overall best to choose grain size & types based on **application domain model**, Example: Users, Orders, etc

External Changing Data

- Grain can re-read current state data from underlying backing storage using `State.ReadStateAsync()`
  This is good way to force "resync" with underlying DB changes.
- Alternately, grain can use a **timer** to re-read data from backing storage periodically,
  based on suitable "staleness" decisions for an application. Example: Content Cache grain.
- Adding / Removing Fields
  - Storage provider in use will determine effects of adding / removing additional fields from persisted state.
  - Due to no-schema, Azure table storage should automatically adjust to extra fields, but best to test thoroughly!

Writing Custom Storage Providers

- Storage providers are a major extensibility point for Orleans, and easy to write.
  - Tutorial: https://orleans.codeplex.com/wikipage?title=Custom%20Storage%20Providers&referringTitle=Step-by-step%20Tutorials
- Storage API contract for grains driven by the GrainState API – `Write/Clear/ReadStateAsync()`
- Storage behavior contract defined by storage provider, typically configurable.
  Example: Batch Write's, Hard vs Soft Delete, etc

Cluster Management

- Orleans automatically manages cluster liveness
  - Worker roles (silo) may fail and join at any time
  - Orleans membership handles all automatically
  - Silo instance table for diagnostics
  - Tunable configuration options: more aggressive vs. more lenient failure detection
- Failures are the norm, can happen any time
  - Lost grains will be automatically reactivated
  - In-process grain calls will fail or timeout
  - Orleans provides best effort message delivery
  - Any network message can be lost, should be retried by application code if important (usual practice is to retry end to end from the client/front end).
- Currently no graceful shutdown
  - Azure upgrade/reboot is treated as node failure

Deployment & Production Management

- Service monitoring
  - Utilize info provided by Orleans
    - Windows perf counters
    - Compact Azure metrics table
    - Very detailed Azure statistics table.
    - Watch for specific log events in Trace
  - Add your own perf counters
- Scaling out and in
  - Monitor your SLA, utilization
  - Add/remove instance
  - Orleans automatically rebalances and takes advantage of the new HW
- Version management
  - No in-place code upgrade, have to restart the silo
  - If the change is backward compatible, can restart silos one by one, e.g. Azure upgrade
  - Otherwise have to restart the whole deployment
    - Azure VIP swap vs. downtime
    - Beware of storage accounts when two deployments are running in parallel
    - Fully stop old deployment before starting the new one or be idempotent

Logging & Testing

- Logging, tracing & monitoring
  - Use GrainBase.GetLogger() that exposes Info(), Warn(), Error(), Verbose()
  - By default output goes to .NET Trace along with Runtime traces, to local file
  - Easy to consume by Windows Azure Diagnostics
  - Can install your own log consumer via Logger.LogConsumer.Add()
  - Can override default trace levels for grains or runtime components;
    - <TraceLevelOverride LogPrefix="Application" TraceLevel="Verbose" />
    - <TraceLevelOverride LogPrefix="Runtime" TraceLevel="Warning" />
- Testing
  - Will publish UnitTestBase class for easy unit testing
    - Starts two (or more) silos in app domains and a client in the main app domain
    - Simplifies version of what we use internally

Troubleshooting

- Logs, logs, logs: silo logs, client (frontend) logs
  - WAD may or may not pick up the logs in case of startup failures
  - RDP to the machines to be sure
- Use Azure Table-based membership for development/testing
  - Works with Azure Storage Emulator for local troubleshooting
  - OrleansSiloInstances table shows state of the cluster
  - Use unique deployment IDs (partition key) for simplicity
- Silo doesn't start
  - Look at OrleansSiloInstances – did the silo register there?
  - Is firewall open for TCP ports 11111 & 30000 (by default, can change in config)?
  - Look at the log, there's an extra one for startup errors
  - RDP to the server or Worker Role instance and try to start it manually
- Client (frontend) can't connect to silo cluster
  - Has to be in the same hosted service as silos
  - Look at OrleansSiloInstances – are there silos (gateways) registered?
  - Look at the client log – does it find gateways listed in OrleansSiloInstances table?
  - Look at the client log – can it connect to one or more gateways?

Questions?