Programmer’s Guide to Orleans

For the Orleans alpha release, April 2012

Table of Contents

[Introduction 2](#_Toc323287701)

[Overview of the Orleans Architecture 2](#_Toc323287702)

[Actors 3](#_Toc323287703)

[Persistence 3](#_Toc323287704)

[Replication for scalability 3](#_Toc323287705)

[Consistency 3](#_Toc323287706)

[Key Concepts 4](#_Toc323287707)

[Orleans and Client Code 4](#_Toc323287708)

[Grains: Units of Distribution 4](#_Toc323287709)

[Grain Interfaces: Service Contracts 5](#_Toc323287710)

[Grain Reference: Service Endpoint 5](#_Toc323287711)

[Promises: Units of Asynchrony 6](#_Toc323287712)

[Turns: Units of Execution 7](#_Toc323287713)

[Transactions: Units of Isolation 8](#_Toc323287714)

[Observers 9](#_Toc323287715)

[Developing a Grain 10](#_Toc323287716)

[Categories of Grains 10](#_Toc323287717)

[Orleans-Managed Grains 10](#_Toc323287718)

[Self-Managed Grains 11](#_Toc323287719)

[Defining the Grain Interface 11](#_Toc323287720)

[Generating the Class Factory 12](#_Toc323287721)

[The Implementation Class 12](#_Toc323287722)

[Persisted State Interface 12](#_Toc323287723)

[Grain Class 13](#_Toc323287724)

[Using the Grain 15](#_Toc323287725)

[Connecting to Orleans 16](#_Toc323287726)

[Connecting to a gateway 17](#_Toc323287727)

[Configuring the Client 17](#_Toc323287728)

[Find or create grains 17](#_Toc323287729)

[Sending messages to grains 18](#_Toc323287730)

[Receiving notifications 18](#_Toc323287731)

[Example 19](#_Toc323287732)

[Persistence 20](#_Toc323287733)

[Production Readiness of Orleans Features 21](#_Toc323287734)

# Introduction

Orleans is a framework for building highly-scalable cloud applications. Its programming model and distributed runtime system assume much of the burden involved in developing, deploying, and operating a highly scalable application running on potentially unreliable servers. Orleans provides a higher level programming model than existing .NET libraries, significantly reducing the amount of work that developers need to do while easing the transition from desktop or client/server systems to cloud-scale distributed applications. Orleans programs can co-exist and communicate with existing applications, but to obtain the benefits of Orleans an application needs to be written using the Orleans programming model. Orleans applications can easily be deployed in Azure or on a private cluster.

This document provides an introduction to Orleans for developers planning to write new applications. It covers the Orleans architecture, describes the key concepts in some detail, and provides basic information on how to build, deploy, and run an Orleans application. Detailed API and configuration information is documented separately in the Orleans distribution.

# Overview of the Orleans Architecture

Orleans has two primary architectural goals:

* Make it easy for developers with no experience in distributed systems to build cloud-scale applications.
* Ensure that those systems scale across multiple orders of magnitude of load without requiring extensive re-design or re-architecture.

In order to meet these goals, we have intentionally constrained the programming model in order to guide developers down a path of best practices leading to scalable applications. In some cases, such as persistence, we have removed an entire aspect from the explicit programming model and left the functionality to the model and runtime, to ensure a scalable approach.

## Actors

Orleans starts with a basic model of actors interacting through asynchronous message passing. Actors, isolated single-threaded components that encapsulate both state and behavior. They are similar to objects, and should be natural to any developer. Asynchronous messaging differs greatly from synchronous method calls, but long experience has shown that purely synchronous systems do not scale – in this case we have traded familiarity for scalability.

In order to avoid confusion with other systems, actors in Orleans are named ***grains***. Messages are represented by special methods on the .NET interface for a grain type. The methods are regular .NET functions, except that they must return a ***promise***, a construct that represents a value that will become available at some future time. Grains are single-threaded and process messages one at a time, so that developers do not need to deal with locking or other concurrency issues.

## Persistence

Most cloud-scale applications operate for an extended period and require long-lived state, also known as persistence. In Orleans, grains encapsulate all state, and so data persistence is grain persistence. Developers specify the persistence behavior of their grain types, such as which properties should be persisted, which properties may be used for finding grains (i.e., querying), and which properties should be unique. The Orleans runtime ensures grain state is transparently persisted and loaded from the persistent store at appropriate times. In this regard, Orleans provides functionality similar to object-relational mapping tools.

## Replication for scalability

Actor-based systems fail to scale if a single actor is overloaded and becomes a bottleneck. Orleans uses actor replication to avoid this situation: if a single grain has many requests, the Orleans runtime will automatically create multiple replicas of the grain. These replicas, called ***activations***, run in parallel, each processing one distinct request at a time. The system creates and destroys activations as needed to handle the current load. Replication is transparent to the developer, which allow to reason about a grain as a single logical entity. Activations are managed by the runtime with no developer involvement.

Each replica independently changes its local replica of the grain’s state as it processes messages. There is no single, shared copy of the state, no master copy, and no locking – these approaches to providing single-copy semantics scale poorly. Instead, the Orleans runtime transparently reconciles changes between different replicas using a multi-master replication algorithm and updates the grain’s persistent state. The grain developer defines the reconciliation policy by selecting the data types used for persistent properties. The default semantics is “last writer wins”, unless the developer uses one of the reconcilable data structures provided by Orleans or creates her own reconcilable data structure with a custom reconciliation procedure.

## Consistency

Replicas introduce the possibility of inconsistency. If an activation of grain 1 sends two messages to grain 2 while processing a request, it would be confusing if the two requests went to two different activations of grain 2, with different internal state. To avoid this, Orleans implements a ***lightweight transaction*** mechanism that assures each external request is handled by the same activation of each grain, ensuring a consistent view of the application’s state for a request. Orleans transactions do not offer full serializability, but they do ensure that each transaction always sees a causally consistent view of the system.

Transactions also automate failure handling: network failures and system crashes cause the active transaction to be rolled back and replayed. This relieves the developer of much of the burden of dealing with system failures, a common source of errors in large-scale applications.

# Key Concepts

Orleans is based on an asynchronous, single-threaded, distributed-object model. The units of distribution, encapsulating data and computation, are called **grains**. Grains interact by passing asynchronous messages, whose return values are represented in the code as **promises**.

## Orleans and Client Code

An Orleans application consists of two distinct parts: the Orleans (grain based) part, and the client part. The Orleans part is comprised of application grains hosted by Orleans Runtime servers called silos. Grain code is executed by the runtime under the scheduling restrictions and guarantees detailed later. The client part connects to the Orleans part via a thin layer of Orleans Client library that enables communication of the client code with grains hosted by the Orleans part via grain references. The client part in this context means a client to the Orleans part, but it can run as part of a client or server applications. For example, an ASP.NET application running on a web server can be a client part of an Orleans application. The client part executes on top of the .NET thread pool, and is not subject to scheduling restrictions and guarantees of the Orleans Runtime.

## Grains: Units of Distribution

Grains are building blocks of an Orleans application. Grains are the atomic units of isolation, distribution, and persistence. A grain encapsulates state and behavior, like any .NET object. The declared state of a grain is persistent; Orleans automatically transfers the grain’s state between server memory and the persistent store, with no application code required. A grain is a logical entity that at any point in time may have zero or more in-memory replicas called **activations**. A grain may exist only in the persistent store with no in-memory replicas if there are no requests pending for the grain. When there is work for the grain, the run-time will create an activation of the grain by picking a server and there instantiating the .NET class that implements the behavior of the grain, initializing the instance with the grain’s persistent state. As the state of the grain is modified within an activation in response to requests made of the grain, the persistent state is transparently synchronized with the in-memory version as application transactions are completed.

Orleans controls the process of activating, synchronizing, and deactivating grains. This process is transparent to the developer: when coding a grain, a developer should assume that any other grains that the current grain will interact with are activated. Similarly, the developer does not need to write code to serialize the state of the grain or to transfer the grain’s state between memory and the persistent store. In particular, Orleans will ensure that changes to grain state are written out to the store in a consistent way by updating the state of all grains involved in an application transactions atomically at the completion of the task; see Transactions: Units of Isolation, below.

Grains are isolated; the only way for two grains to interact is by sending messages. They have no shared memory or other shared state. In addition, grain activations are single-threaded. While many activations of the same grain may execute concurrently, each activation executes at most one logical unit of work, known as a **turn**, at a time. This means that there is no need to use locks or other local synchronization mechanisms in grain code.

In this alpha release, Orleans supports two modes: *single activation* mode (default), in which only one simultaneous activation of every grain is created, and *stateless* *worker* mode, in which independent activations of a grain are created to increase the throughput. “Independent” specifically implies that there is no state reconciliation between different activations of the same grain, so this mode is appropriate for grains that hold no local state, other than perhaps cached data from some other store.

In future releases, Orleans will support more advanced multiple activation modes to provide scale-out and fault-tolerance. Supporting multiple activations will require some additional application declarations to specify merge behavior when different modifications to a grain’s state are made in different activations; this will be documented out in a future release of Orleans.

## Grain Interfaces: Service Contracts

Grains interact with each other by invoking methods and properties declared as part of the respective grain interfaces. A grain implements one or more previously declared grain interfaces. All methods and properties of a grain interface are required to be asynchronous, that is their return types have to be promises (see Promises: Units of Asynchrony for more details on promises).

Example:

1. public interface IChirperPublisher : IGrain
2. {
3. AsyncValue<long> UserId { get; }
4. AsyncValue<string> UserAlias { get; }
5. AsyncValue<string> DisplayName { get; }
6. AsyncValue<List<ChirperMessage>> GetPublishedMessages(int n = 10,
7. int start = 0);
8. AsyncCompletion AddFollower(string userAlias,
9. IChirperSubscriber follower);
10. AsyncCompletion RemoveFollower(string userAlias,
11. IChirperSubscriber follower);
12. }

## Grain Reference: Service Endpoint

A grain reference is a logical endpoint that allows other grains, as well as non-grain client code, to invoke methods and properties of a particular grain interface implemented by a grain. A grain reference is a proxy object that implements the corresponding grain interface. A grain reference can be obtained by creating a new grain, looking up an existing grain, or as the return value of a method or property. A grain reference can be passed as an argument to a method call.

## Promises: Units of Asynchrony

Grains interact by sending asynchronous messages. As in most modern programming models, these messages are exposed as method calls. Unlike traditional RPC models, however, these method calls return immediately with a **promise** for a future result, rather than blocking until the result is returned. This resolves the impedance mismatch between synchronous method calls and asynchronous message passing. Promises allow for concurrency without requiring explicit thread management.

Promises have a simple lifecycle. Initially, a promise is **unresolved**; it represents the expectation of receiving a result at some unspecified future time. When the result is received, the promise becomes **fulfilled** and the result becomes the value of the promise. If an error occurs, either in the calculation of the result or in the communications between the requesting and responding grains, then the promise becomes **broken**, and has no value. A promise that has been fulfilled or broken is considered **resolved**.

The primary way to use a promise is to schedule a closure (also referred to as continuation) to execute when the promise is resolved. Closures are scheduled by calling the ContinueWith() method on a promise. The closure executes in its own turn (see Turns: Units of Execution below), so no more than one such closure (per grain activation) will ever execute simultaneously. Because promises are resolved asynchronously, the order in which the closures for different promises run is not predictable. This interleaving of turns for an activation will never result in a fine-grained data race, but does introduce concurrency issues similar to cooperative multitasking: since the order in which promises resolve is not predictable, the state of the activation when a closure is executed may be different than its state when the closure was first scheduled on the promise.

Example:

1. IChirperPublisher userToFollow =
2. ChirperPublisherFactory.LookupUserAlias(alias);
3. AsyncCompletion continuationPromise =
4. userToFollow.ContinueWith((userId) =>
5. {
6. return FollowUser(userId, alias, userToFollow);
7. });

It is also possible to join two or more promises; that is, to create a new promise that is resolved when all of its constituent promises are fulfilled, or when any one is broken. This is a useful pattern when a grain needs to start multiple computations and then wait for all of them to complete before proceeding. For example, a front-end grain that generates a web page made of many parts might make multiple back-end calls, one for each part, and receive a promise for each result. The grain would then wait for the join of all of these promises; when the join is resolved, the individual promises have been fulfilled, and all the data required to format the web page has been received.

Example:

1. List<AsyncCompletion> promises = new List<AsyncCompletion>();
2. ChirperMessage chirp = CreateNewChirpMessage(text);
3. foreach (IChirperSubscriber subscriber in Followers.Values)
4. {
5. promises.Add(subscriber.NewChirp(chirp));
6. }
7. AsyncCompletion joinedPromise = AsyncCompletion.JoinAll(promises);

Finally, the Orleans programming model allows a method to suspend execution and wait for a promise to resolve. In this case, the waiting turn is ended, and the method invocation continues in a new turn when the promise resolves. Waiting takes up additional resources over scheduling a continuation and so is advised against as explained below.

Orleans provides both completion promises and value promises. Completion promises correspond to void methods in normal object-oriented programming: the promise is resolved when the requested operation is completed, but conveys no information other than that the request succeeded. Completion promises are instances of the AsyncCompletion class.

Value promises correspond to non-void methods. A value promise represents the eventual expectation of a value of a specified type; this value is made available to closures attached to the promise. Value promises are instances of the AsyncValue<T> generic class.

Promises for a reference to a grain are handled as a special case by Orleans. Methods that return a grain reference return the grain interface type directly, rather than an AsyncValue<IGrainType>, as if the reference was already fulfilled. This allows **pipelining**: methods may be invoked on a grain interface whether it is resolved or not. An unresolved grain reference internally queues up requests and sends them once the reference is resolved.

In the near future we plan to integrate promises with the .NET 4.5 async/await functionality to make writing asynchronous code even easier.

## Turns: Units of Execution

As mentioned above, a grain activation performs work in chunks and finishes each chunk before it moves on to the next. Chunks of work include method invocations in response to requests from other grains or external clients, and closures scheduled on resolution of a promise. The basic unit of execution corresponding to a chunk of work is known as a **turn**.

While Orleans may execute many turns belonging to different activations in parallel, each activation will always execute its turns one at a time. This means that there is no need to use locks or other synchronization methods to guard against data races and other multithreading hazards. As mentioned above, however, the unpredictable interleaving of turns for scheduled closures can cause the state of the grain to be different than when the closure was scheduled, so developers must still watch out for interleaving bugs.

Every promise continuation constitutes a separate turn. In addition, if grain code blocks on a promise using the promise’s Wait() method, the current turn is ended. When the promise is resolved, a new turn starts that executes the remainder of the method. Other turns for the same activation may execute while the one routine is suspended in the Wait call, and thus the state of the activation (including the values of activation properties and fields) may change between the invocation of the Wait method and the return from the Wait. In the following example, even though p2.Wait() blocks execution of the method, Continuation1() may (or may not) execute and change state of the activation *before* p2.Wait() returns. For this reason (and due to inefficient usage of system resources), the use of Wait is discouraged.

Example:

1. AsyncCompletion p1 = Operation1();
2. AsyncCompletion p2 = Operation2();
3. AsyncCompletion p3 = p1.ContinueWith(() => Continuation1());
4. p2.Wait();

By default, Orleans requires an activation to completely finish processing one request before accepting the next request. An activation will not accept a new request until all of the promises created (directly or indirectly) in the processing of the current request have been resolved and all of their associated closures executed. Grain implementation classes may be marked with the Reentrant attribute to indicate that turns belonging to different requests may be freely interleaved. In other words, a reentrant activation may start executing another request while a previous request has not finished processing yet and has pending closures. Execution of turns of both requests will still be limited to a single thread.

## Transactions: Units of Isolation

Completing one request before starting a new request prevents different requests to the same grain from interfering with each other, but it is not enough to guarantee that a client grain will see a consistent view of a grain it interacts with multiple times. In particular, a grain could send a message to another grain setting a property, and then send a message reading the value of that property, and see a different value than it wrote.

This inconsistency could occur in two different ways: the two messages could be delivered to two different activations of the same grain, or the two messages could be delivered to the same activation but another grain could have modified the property in between the writing and reading messages. The first is a failure of consistency: the client grain is interacting with different, inconsistent replicas of the same server grain. (In this alpha release Orleans only creates a single activation of each grain, so this inconsistency cannot occur.) The second is a failure of isolation: operations from multiple logical requests are interleaving, and thus interfering, at the server grain.

Orleans provides a mechanism to prevent this type of inconsistency and provide grains with a consistent, isolated view of their environment. When a new request enters the Orleans system from outside, a **transaction** is created as context for processing the request. As grain activations participating in the transaction make new requests, the transaction context flows with these requests and the targeted activations are joined to the transaction. When the initial request is completed, the transaction completes, releasing the participating activations. The transaction then commits by updating persistent state in a single indivisible operation, ensuring atomicity and durability.

Consistency and isolation are ensured by two behaviors: within a transaction, all requests for a grain are routed to the same activation, ensuring a consistent view of its state; and, once an activation is joined to a transaction, it does not accept any requests from a different transaction, ensuring isolation. Essentially, the transaction is associated with a lazily-formed, consistent, isolated snapshot of the system.

Transactions provide failure atomicity as well. When a system error such as a failed server occurs during the processing of a transaction, Orleans rolls back the entire transaction and restarts it from the beginning. Any state changes made during the failed invocation are discarded. It is important to note that the restarted execution will in general build a different consistent snapshot of the system, and so might not produce the same results as the earlier execution.

For transactions that do not modify the state of the grain, the default behavior is too conservative, providing more guarantees than necessary with concomitant unnecessary costs. Methods that start read-only transactions may be marked with the ReadOnly attribute. This allows Orleans to optimize the execution of the transaction; in particular, a single activation may participate in multiple read-only transactions simultaneously.

For some applications, the guarantees provided by the transaction system are unnecessary. In this case, the transaction system may be turned off in the system configuration. For example, for an application that performs a partitioned bulk-loading phase and then a read-only querying phase, the partitioning by itself may provide sufficient isolation for the bulk loading, and running the application without the transaction system may improve the performance of the bulk-loading phase.

In this alpha release, the transaction system is still in the *proof of concept* state and is turned off by default.

## Observers

Because the standard .NET event facility is explicitly synchronous[[1]](#footnote-1), it doesn’t fit into an asynchronous framework such as Orleans. Instead, Orleans uses the Observer pattern.

A grain type that allows observation will define an observer interface that inherits from the IGrainObserver interface. Methods on an observer interface correspond to events that the observed grain type makes available. An observer would implement this interface and then subscribe to notifications from a particular grain. The observed grain would call back to the observer through the observer interface methods when an event has occurred.

Methods on observer interfaces must be void since event messages are one-way. If the observer needs to interact with the observed grain as a result of a notification, it must do so by invoking normal methods on the observed grain.

The observed grain type must expose a method to allow observers to subscribe to event notifications from a grain. In addition, it is usually convenient to expose a method that allows an existing subscription to be cancelled. Grain developers may use the Orleans ObserverSubscriptionManager<T> generic class to simplify development of observed grain types.

# Developing a Grain

In this section we walk through the steps involved in defining and using a new Chirper Account grain type as used in the Chirper sample application in the Orleans SDK. The grain type we define will have three properties, a user name (alias), a user id, and a display name. A method for adding / removing a new subscriber, and retrieving a list of previously published messages (known as “Chirps”) is also included. Individual grains will be indexed by the user id or user alias and searchable by display name.

We will create three separate pieces of code: the grain interface definition, the grain implementation, and a standard C# class that uses the grain. Each of these belongs in a different project, built into a different DLL: the interface needs to be available on both the “client” and “server” sides, while the implementation class should be hidden from the client, and the client class from the server. The interface project should be created using the Visual Studio “Orleans Grain Interface Project” template that is included in the Orleans SDK, and the grain implementation project should be created using the Visual Studio “Orleans Grain Implementation Class Project” template. The grain client project can use any standard .Net code project template, such as the standard Console Application or Class Library templates.

## Categories of Grains

Orleans supports two categories of grains: Orleans-managed (normal) grains and self-managed grains. The biggest difference between the two categories lies in the life cycle of the grains. Orleans-managed grains require a connection to a SQL server for persisting the grain index (for a production environment; for development there is a built-in in-memory index - DomainIndexGrain). An application that only uses self-managed grains does not have a dependency on SQL server because self-managed grains are not indexed.

### Orleans-Managed Grains

Normal grains are explicitly created and deleted by calling the CreateGrain() and Delete() static methods on their factory classes. In between these two calls a grain exists in the system, even if in a dehydrated state with no in-memory activation. Orleans keeps the fact of the grain’s existence in its index, and allows the application code to retrieve the grain (get a grain reference for it) by querying on one or more of its queryable properties. Orleans manages the persistent state of queryable properties of normal grains, and other properties if specified.

### Self-Managed Grains

Self-managed grains cannot be explicitly created or deleted. They always exist “virtually”, and get activated when a request is sent to them. Self-managed grains cannot have queryable properties and cannot be looked up or queried for. In the current version of Orleans, self-managed grains have a long integer identity within a grain type. Application code “manages” the activation of a self-managed grain by calling the GetGrain(long id) static factory method for a specific grain identity to obtain a grain reference for it, and then sending a message to it. A self-managed grain interface must inherit from ISelfManagedGrain instead of IGrain, and there must be only one concrete grain class for each grain interface. The long integer identity of a self-managed grain can later be retrieved via ISelfManagedGrain.GetPrimaryKey() extension method.

## Defining the Grain Interface

A grain type is defined by an interface that inherits from the Orleans.IGrain interface for regular, Orleans-managed, grains, or from ISelfManagedGrain for self-managed grains.

All of the methods in the grain interface must return a promise (Orleans.AsyncCompletion, Orleans.AsyncValue<T>, or a descendant of Orleans.IGrain). The underlying type T for AsyncValue<T> value promises must be serializable.

Grain interfaces may define properties, but may not define property setters because the .Net property setting notation is synchronous[[2]](#footnote-2). Property values must be promises, and the underlying type must be serializable.

Properties in the interface of a regular grain may be used as search keys to find grains by content Properties that can be used in a query should be marked with the Orleans.Queryable attribute, which causes the Orleans runtime to automatically create an index for that property. A property that provides a unique primary key for the grain type should be marked with the Orleans.Queryable(IsUnique=true) attribute. A grain interface may contain multiple unique primary key properties. Grain types need not support content-based search, and so need not define any Orleans.Queryable properties.

1. public interface IChirperPublisher : Orleans.IGrain
2. {
3. [Orleans.Queryable(IsUnique = true)]
4. AsyncValue<long> UserId { get; }
5. [Orleans.Queryable(IsUnique = true)]
6. AsyncValue<string> UserAlias { get; }
7. [Orleans.Queryable] AsyncValue<string>
8. DisplayName { get; }
9. [Orleans.ReadOnly]
10. AsyncValue<List<ChirperMessage>>
11. GetPublishedMessages(int n = 10, int start = 0);
12. AsyncCompletion AddFollower(string alias, IChirperSubscriber subs);
13. AsyncCompletion RemoveFollower(string alias, IChirperSubscriber subs);
14. }

In the example, note the use of the Queryable and IsUnique attributes to indicate that both the UserAlias and UserId properties are searchable unique keys (lines (3) and (6)), while the DisplayName property (line (9))is a searchable key but values will not necessarily be unique.

The GetPublishedMessages method is marked with the Orleans.ReadOnly attribute (line (12)) to indicate that this method does not modify the grain’s state. That method will return an asynchronous promise for a List of ChirperMessage data objects.

The AddFollower (line (15)) and RemoveFollower (line (16)) methods take two parameters: a user alias string and an IChirperSubscriber grain reference for the follower to add or remove. The Add/RemoveFollower methods both return Orleans.AsyncCompletion – which is the asynchronous promise equivalent of void.

## Generating the Class Factory

After the grain interface has been defined, building the project originally created with the Orleans Visual Studio project template will use the Orleans specific MSBuild targets to generate client proxy and factory classes corresponding to the grain interfaces user defines, and merge this additional code back into the interface DLL. (The code generation tool –ClientGenerator.exe - can also be invoked directly as a part of post build processing. However this should be used with caution and is generally not recommended.)

The most important class in the generated proxy code is the grain factory class, which is named after the grain interface by stripping off the initial “I” and appending “Factory”. For instance, if your grain interface is IChirperAccountGrain, then your grain factory class will be called ChirperAccountGrain***Factory***. The namespace for this class has the same name as the namespace that the interface is in.

## The Implementation Class

A grain type is implemented by a class that implements the grain type’s interface and inherits directly or indirectly from Orleans.GrainBase.

### Persisted State Interface

If the grain class needs Orleans to manage persistence of its state, the grain class needs to inherit from GrainBase<T>, where T is the persisted state interface that defines a list of properties to be managed by Orleans, which can be viewed as a contract between the grain class and Orleans. T must derive from IGrainState marker interface. This is one of the major changes in the April release. Prior to that, the persistence contract was implied through public properties on the grain interface itself, and fulfilled through a matching set of prompt properties in the generated base class. Starting with the April release, there are no generated base classes and GrainBase<T> exposes a protected property State of type T that is the explicit implementation of the persisted state interface provided by Orleans to the grain class.

The Chirper sample defines IChirperAccountState as the persisted state interface for ChirperAccount grain.

1. public interface IChirperAccountState : IGrainState
2. {
3. SyncDictionary<ChirperUserInfo, IChirperPublisher> Subscriptions { get; set; }
4. SyncDictionary<ChirperUserInfo, IChirperSubscriber> Followers { get; set; }
5. int ReceivedMessagesCacheSize { get; set; }
6. int PublishedMessagesCacheSize { get; set; }
7. SyncFixedQueue<ChirperMessage> RecentReceivedMessages { get; set; }
8. SyncFixedQueue<ChirperMessage> MyPublishedMessages { get; set; }
9. long UserId { get; }
10. string UserAlias { get; }
11. [Orleans.Queryable]
12. string DisplayName { get; }
13. }

### Grain Class

The ChirperAccount grain class implements three interfaces: IChirperPublisher, IChirperSubscriber and IChirperAccount. For the purposes of this example, we will focus on the implementation of the IChirperPublisher grain interfaces. See the Chirper sample application for the complete implementation.

Interface methods that return an AsyncCompletion should return AsyncCompletion.Done (lines (36) and (48)). Properties and methods that return value promises may return an explicit value, which will be implicitly cast to an AsyncValue (line (45)), or an explicit promise.

1. namespace Orleans.Samples.Chirper.Grains
2. {
3. public class ChirperAccount : GrainBase<IChirperAccountState>, IChirperAccount
4. {
5. #region GrainBase overrides
6. public override void SetDefaults()
7. {
8. base.SetDefaults();
9. State.PublishedMessagesCacheSize = 100;
10. State.MyPublishedMessages = new SyncFixedQueue<ChirperMessage>(State.PublishedMessagesCacheSize);
11. State.Followers = new SyncDictionary<ChirperUserInfo, IChirperSubscriber>();
12. }
13. #endregion
14. #region IChirperPublisher interface methods
15. public override AsyncValue<List<ChirperMessage>>
16. GetPublishedMessages(int n, int start)
17. {
18. if (start < 0) start = 0;
19. if ((start + n) > State.MyPublishedMessages.Count) n = State.MyPublishedMessages.Count - start;
20. return State.MyPublishedMessages.Skip(start).Take(n).ToList();
21. }
22. public override AsyncCompletion AddFollower(string alias,
23. IChirperSubscriber follower)
24. {
25. ChirperUserInfo userInfo = ChirperUserInfo.GetUserInfo(userId, alias);
26. if (State.Followers.ContainsKey(userInfo))
27. {
28. State.Followers.Remove(userInfo);
29. }
30. State.Followers[userInfo] = follower;
31. return AsyncCompletion.Done;
32. }
33. public override AsyncCompletion RemoveFollower(string alias,
34. IChirperSubscriber follower)
35. {
36. IEnumerable<KeyValuePair<ChirperUserInfo, IChirperSubscriber>> found = State.Followers.Where((f) => f.Key.UserAlias == alias);
37. if (found.Count() > 0)
38. {
39. ChirperUserInfo userInfo = found.FirstOrDefault().Key;
40. State.Followers.Remove(userInfo);
41. }
42. return AsyncCompletion.Done;
43. }
44. #endregion
45. }
46. }

## Using the Grain

Once we have our grain type implemented, we can write a client application that uses the type.

The following Orleans DLL libraries from either the [SDK-ROOT]\Binaries\OrleansClient or [SDK-ROOT]\Samples\References directories will need to be referenced in the client application project:

* Orleans.dll

Almost any client will involve use of the grain factory class, mentioned above. There are several important methods exposed by grain factories.

For Orleans-managed grains:

* The CreateGrain method is used to create a new grain. It optionally allows the properties of the new grain to be specified. It also allows a collection of GrainStrategy objects to be specified; this is generally not required. This method returns a (promise for) a reference to the new grain.
* The Delete method is used to delete an existing grain. It returns a completion promise.
* The Lookup method is used to find a grain by primary key. If the grain type does not define a property with the Queryable(IsUnique = true) attribute, then this method will not be available. This method returns a (promise for) a grain reference.
* The Where method is used to find a set of grains that match a LINQ-style query expression. If the grain type does not define any Queryable attributes, then this method will not be available. This method returns a promise for a list of grain references.

For self-managed grains:

* The GetGrain method is used for getting a grain reference for a particular ID. As already mentioned above, self-managed grains cannot be explicitly created or deleted.

1. namespace SimpleChirperClient
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. OrleansClient.Initialize();
8. String alias = args[0];
9. IChirperAccount account =
10. ChirperAccountFactory.LookupUserAlias(alias);
11. AsyncCompletion promise = account.GetPublishedMessages(10)
12. .ContinueWith(
13. (List<ChirperMessage> chirps) =>
14. {
15. foreach (ChirperMessage chirp in chirps)
16. {
17. Console.WriteLine(string.Format(
18. @"New chirp from @{0} at {1} on {2}: {3}",
19. chirp.PublisherAlias,
20. chirp.Timestamp.ToShortTimeString(),
21. chirp.Timestamp.ToShortDateString(),
22. chirp.Message));
23. }
24. }),
25. (Exception exc) =>
26. {
27. Console.WriteLine("Error connecting Chirper client for user="
28. + alias + " – " + exc.ToString());
29. });
30. promise.Wait();
31. }
32. }
33. }

The above code demonstrates a few useful patterns for dealing with promises:

* Processing of returned data is chained together using ContinueWith closures to schedule follow-on processing even before the original lookup call has necessarily completed.   
  The sample code above will cause several things to happen:
  + Lookup of ChirperAccount grain with the specified alias (username) [Line (11)]
  + Retrieve the list of the 10 most recently published chirp messages for that user [Line (14)]
  + Process each returned chirp message by printing it to the console window [Lines (15) - (26)]
* An exception clause [Lines (28) - (32)] will handle any broken promise errors from the chain of three promises above (LookupUserAlias on Line (11), GetPublishedMessages on Line (14), and ContinueWith on Line (15)). This allows error handling and recovery to be placed in one place, but respond to an arbitrarily complex chain of asynchronous operations.
* The main program thread performs a rendezvous with the asynchronous processing flow by using an explicit Wait operation before it exists. In general, explicit wait’s should be avoided if possible (use ContinueWith’s instead to schedule follow-on processing) but in this case it is required to prevent the main program thread existing before the async processing is completed.

See the Key Concepts section for more details on the various ways to use asynchronous promises for execution scheduling and exception flow.

# Connecting to Orleans

To allow applications to communicate with grains from outside Orleans, the framework includes a **client library**. This client library might be used by a desktop or mobile application, or it might be used by a server application that renders interactive web pages or exposes a web services API. The client library provides a subset of the Orleans programming model for writing asynchronous clients that can find, create, and communicate with Orleans grains. This requires a few simple steps:

1. Connect to an Orleans gateway
2. Find existing grains or create new ones
3. Send messages to grains and receive responses
4. Receive asynchronous notifications from grains via observers

## Connecting to a gateway

To establish a connection, a client calls OrleansClient.Initialize(). This will connect to the gateway silo at the IP address and port specified in the ClientConfiguration.xml file. This file must be placed in the same directory as the Orleans.dll library used by the client. As an alternative, a configuration object can be passed to OrleansClient.Initialize programmatically instead of loading it from a file.

### Configuring the Client

In ClientConfiguration.xml, the Gateway element specifies the address and port of the gateway endpoint that need to match those in OrleansConfiguration.xml on the silo side:

<ClientConfiguration xmlns="urn:orleans">

  <Gateway Address="<IP address or host name of silo>" Port="30000" />  
</ClientConfiguration>

#### Configuring Silos

In OrleansConfiguration.xml, the ProxyingGateway element specifies the gateway endpoint of the silo, which is separate from the inter-silo endpoint defined by the Networking element and must have a different port number:

 <?xml version="1.0" encoding="utf-8"?>

<OrleansConfiguration xmlns="urn:orleans">

<Defaults>

<Networking Address="" Port="11111" />

<ProxyingGateway Address="" Port="30000" />

</Defaults>

</OrleansConfiguration>

## Find or create grains

After establishing a connection, the client library state is initialized to enable the static methods in the generated factory classes to find or create grains, such as ChirperAccountFactory.LookupUserAlias(string UserAlias) or ChirperAccountFactory.CreateGrain().

If you are not sure if the user account exists or not, and it does not require any initial properties to be provided on creation, you can use a single request to either lookup a grain if it exists, or create one if it does not:

ChirperAccountFactory.LookupUserAliasOrCreate(string UserAlias)

For self-managed grains, all you need to do is call the GetGrain factory method and pass it a long integer ID of the grain:

MyGrainFactory.GetGrain(lond id)

In the current alpha SDK release, the gateway silo does not perform any client authentication or authorization, and the runtime provides query access to all grains in the system. Access to Orleans should be controlled by network isolation so only authorized clients can connect to Orleans. Application-level authentication and authorization must be performed in the client or in application grains. A future release will allow Orleans applications to use security tokens provided to the gateway to limit the capability of the client’s grains.

## Sending messages to grains

The programming model for communicating with grains from a client is almost the same as from a grain. The client holds grain references which implement a grain interface like IChirperAccount. It invokes methods on that grain reference, and these return asynchronous values: AsyncCompletion, AsyncValue<T>, or another grain interface inheriting from IGrain. The client can use ContinueWith() to queue continuations to be executed when these asynchronous values resolve, or Wait() to block the current thread. One key difference is that grains are constrained to be single-threaded by the Orleans scheduler, while clients may be multi-threaded. The client library uses the TPL thread pool to manage continuations and callbacks, and so it is up to the client to manage its own concurrency using whatever synchronization constructs are appropriate for its environment – locks, events, TPL tasks, etc.

## Receiving notifications

There are some situations in which a simple message/response pattern is not enough, and the client needs to receive asynchronous notifications. For example, a user might want to be notified when a new message has been published by someone that she is following.

A grain that publishes such notifications provides an API to add or remove observers. An observer is a one-way asynchronous interface that inherits from IGrainObserver, and all its methods must be void. The grain sends a notification to the observer by invoking it like a grain interface method, except that it has no return value, and so that grain cannot depend on the result. The Orleans runtime will ensure one-way reliable delivery of the notifications.

To subscribe to a notification, the client must first create a local C# object that implements the observer interface. It then calls a static method on the observer factory, CreateObjectReference(), to turn the C# object reference into a grain reference, which can then be passed to the subscription method on the notifying grain.

This model can also be used by other grains to receive asynchronous notifications – the subscribing grain simply implements the observer interface as a facet, and passes in a reference to itself, cast to the appropriate observer type (e.g. ChirperViewerFactory.Cast(this.AsReference())).

## Example

Here is a simple example of some client code that connects to Orleans, looks up a chirper account, follows some other accounts, and prints out notifications until the program is manually terminated.

1. public class ChirpPrinter : IChirperViewer
2. {
3. #region Implementation of IChirperViewer
4. public void NewChirpArrived(ChirperMessage chirp)
5. {
6. Console.WriteLine("{0}: {1}", chirp.PublisherDisplayName,
7. chirp.Message);
8. }
9. public void SubscriptionAdded(ChirperUserInfo following)
10. {
11. }
12. public void SubscriptionRemoved(ChirperUserInfo notFollowing)
13. {
14. }
15. #endregion
16. }
17. public static void Main(string[] args)
18. {
19. // connect to the gateway and initialize the client
20. OrleansClient.Initialize();
21. if (args.Length == 0)
22. {
23. Console.WriteLine("usage: chirper MyAccountAlias [FollowAlias ...]");
24. return;
25. }
26. IChirperAccount myAccount =
27. ChirperAccountFactory.LookupUserAlias(args[0]);
28. if (args.Length > 1)
29. {
30. // follow specified users
31. IEnumerable<AsyncCompletion> done = from follow in args.Skip(1)
32. select
33. myAccount.FollowUserAlias(follow);
34. AsyncCompletion.JoinAll(done.ToArray()).Wait();
35. }
36. // create an observer to print messages
37. ChirpPrinter printer = new ChirpPrinter();
38. IChirperViewer observer =
39. ChirperViewerFactory.CreateObjectReference(printer);
40. // make sure not to release the reference to the observer object to
41. // prevent it from being garbage collected prematuraly.
42. // attach it and then go into an infinite loop...
43. myAccount.ViewerConnect(observer).Wait();
44. while (true)
45. {
46. }
47. }

## Persistence

For applications that have straightforward requirements for persistence, Orleans can save and load the persistent state of grains from either files or Azure table storage. The state of a grain will be saved at the completion of every request. If a grain is activated or the deployment is restarted, grain state will be restored from persistent storage. This does not require transactions to be enabled.

To save state in Azure Table Storage, use the following settings in the OrleansConfiguration.xml file:

<Persistence Type="AzureTable" StorageAccount="connection string" />

For development, the system can also save and load grain state to files using the following settings:

<Persistence Type="File" Path="directory path" />

A grain class explicitly defines its persistence contract by inheriting from GrainBase<T> where T is the persistent state interface derived from IGrainState that defines a set of properties to be persisted. This is a significant change from the previous versions of the SDK where persistent state was implicitly derived from public properties of the grain’s interface. A grain class that derives from GrainBase<T> inherits a protected property State of type T that is its conduit for reading and updating its persistent state. Orleans automatically persists grain’s state to the store defined in OrleansConfiguration.xml at the end of each call to a method/property of the grain no marked with a [ReadOnly] attribute.

Here’s an example of a grain with persistent state defined.

1. public interface IRowGrainState : IGrainState
2. {
3. string Sku { get; }
4. string Name { get; set; }
5. int Quantity { get; set; }
6. double Price { get; set; }
7. int InventoryOnHand { get; set; }
8. }
9. public class RowGrain : GrainBase<IRowGrainState>, IRowGrain
10. {
11. AsyncValue<string> IRowGrain.Sku { get{ return State.Sku;}}
12. AsyncValue<string> IRowGrain.Name { get{ return State.Name;}}
13. AsyncValue<int> IRowGrain.Quantity { get{ return State.Quantity;}}
14. AsyncValue<double> IRowGrain.Price { get { return State.Price;}}
15. AsyncCompletion IRowGrain.SetName(string name)
16. {
17. State.Name = name;
18. return AsyncCompletion.Done;
19. }
20. AsyncCompletion IRowGrain.SetQuantity(int quantity)
21. {
22. State.Quantity = quantity;
23. return AsyncCompletion.Done;
24. }
25. AsyncCompletion IRowGrain.SetPrice(double price)
26. {
27. State.Price = price;
28. return AsyncCompletion.Done;
29. }
30. }

# Production Readiness of Orleans Features

While reading this document, it is important to understand that the various features of Orleans are at different level of readiness, and to distinguish between features that have been running in production powering services of a top tier game title, and those that are promising proofs of concepts tested under light load. Because of our shared source approach to building Orleans, we are very open to other people’s and team’s contributions in further development and scale testing of its features.

Fully implemented and tested or running in production at scale:

* Messaging
* Promises, threading and reentrancy guarantees
* Silo reliability: failure detection and recovery
* Orleans-managed grains
* Self-managed grains
* Single activation grain
* Multiple independent activation grains
* End-of-request persistence

Proof of concept features:

* Transactions
* Multiple activations of grains with automatic state synchronization
* General managed persistence of grain state

1. Neither subscription (+=) nor un-subscription (-=) provide a way for an asynchronous result to be returned, [↑](#footnote-ref-1)
2. There’s no mechanism to return an AsyncCompletion from an assignment. [↑](#footnote-ref-2)