# What’s New in Orleans?

## **August Preview Update**

The preview of Orleans that was released in April 2014 has undergone some changes. Most of the modifications are in the form of bug fixes, some reported by preview users (such as a bug in the Reminder implementation), and some that we found ourselves.

We’ve taken feedback from developers internal to Microsoft, and feedback from external users. Not all the feedback has been incorporated yet, but some of it has.

Some of the changes *are going to* break existing code that you may have. There’s no ‘may’ here, it *will* break. We also have a few new features added, which you will hopefully find useful or at least interesting enough to give us feedback on.

**Properties in Grain Interfaces**

A basic .NET API design guideline is that property implementations should not do I/O, and since using grain interfaces typically lead to network I/O, properties do not belong in such interfaces, and Orleans no longer support them. Fortunately, fixing code that depends on this is straight-forward, simply changing the property getter (setters were not supported previously, anyway) to a function:

For example:

public interface IGrain1 : Orleans.IGrain

{

Task<string> SayHello(string greeting);

Task<int> Count { get; }

}

would have to be changed to:

public interface IGrain1 : Orleans.IGrain

{

Task<string> SayHello(string greeting);

Task<int> GetCount();

}

**Public Type Names**

We determined that a lot of type names in the April release were not as descriptive as they could be, and many did not conform to common naming guidelines. We have corrected some of these, but not all.

The most consequential of these is probably ‘GrainBase,’ which was renamed ‘Grain.’ This applies to both the non-generic and the generic variants of the type. This is the change that is guaranteed to affect all Orleans code bases.

We also determined that some types were not needed, and so we removed them. For example, when you create a timer using the April release, you get back an IOrleansTimer reference, which is just an IDisposable with no additional functionality, so it was removed and you now get just an IDisposable.

**Code Generation**

Code generation has changed, mostly simplifications. While there used to be two approaches to code generation, one for assemblies with grain interfaces, one for assemblies with grains, there is now just one. Unfortunately, it means that you have to add an (empty) file ‘orleans.codegen.cs’ under the ‘Properties’ folder of any grain assemblies that you have. Editing the project files using notepad is probably the most expedient way to accomplish this.

On the positive side, though, VB and F# code generation has been improved. In the case of VB, you can now also define grain interfaces using VB. In the case of F#, you can now use F# to define grains with persisted state, which was challenging before.

**Namespaces**

An issue (which, by the way, we knew about in April, but didn’t have time to address) was that the Orleans namespaces weren’t very well planned. Most public types were found in the root namespace, ‘Orleans,’ which did not help developers find the most common types as quickly as we would like.

Therefore, a major overhaul of the namespaces has been done. Most of the types used by developers (IGrain, Grain, IGrainState, IRemindable, etc.) remaining in the root namespace, but many of the secondary types are in lower-level namespaces.

For example, many of the attributes used to control concurrency (e.g. ReentrantAttribute, ImmutableAttributes) are now in the ‘Orleans.Concurrency’ namespace. Placement-controlling attributes are in ‘Orleans.Placement,’ while most other types that you would be like to need are now in ‘Orleans.Runtime.’

Many of these you don’t have to worry about if you are consistently using ‘var’ when declaring local variables. In most other cases, adding a ‘using Orleans.Runtime’ at the top of a file with errors will help.

**New Factory Methods**

In the April preview, the factory methods for remote grain references were code-generated and you would find them by simple pattern matching.

For example, to create a reference to a grain of this interface:

public interface IGrain1 : Orleans.IGrain

you would use the factory method ‘GetGrain’ on a static class created by the compiler:

gref = Grain1Factory.GetGrain(0);

This method is still available (but it may go away based on your feedback), but we have added another way, which is not directly dependent on code generation. Instead, it relies on further specification of the grain as having either a GUID key, an integer key, or a string key. This is done by using one of three new interfaces in place of ‘IGrain’ when declaring a grain interfaces:

public interface IGrainWithGuidKey : IGrain

public interface IGrainWithIntegerKey : IGrain

public interface IGrainWithStringKey : IGrain

For example, IGrain1, which uses an integer key (but there’s currently no type safety around it), would be declared this way:

public interface IGrain1 : Orleans. IGrainWithIntegerKey

Doing so will allow the following to be used to create a grain reference:

gref = GrainFactory.GetGrain<IGrain1>(0);

Note that the new factory methods cannot be used for grain interfaces using ‘IGrain.’ Also note that the new methodology does not allow you to use an extended primary key, i.e. a tuple of GUID/Int64 and string.

We strongly encourage your feedback on this new way of doing things. We think the new methodology is more readable, more aligned with most other frameworks, and we think the type safety is valuable, but what matters are *your* thoughts.

**Non-Azure System Storage**

In the April release, any reliable production-style deployment required using an Azure storage account to keep system state, specifically Orleans cluster status and the data used for the reminders functionality. In the August release, we added SQL Server as a possible location for that data, and this has impacted the server-side configuration.

If the server configuration file used to contain elements like this:

<Globals>  
 <Liveness LivenessType ="AzureTable" />

<Azure DeploymentId="..." DataConnectionString="..."/>

</Globals>

It should now be:

<Globals>  
 <SystemStore SystemStoreType ="AzureTable"

DeploymentId="..."

DataConnectionString="..." />

</Globals>

If, instead, you want to use SQL Server, the configuration should look like this:

<Globals>  
 <SystemStore SystemStoreType ="SqlServer"

DeploymentId="..."

DataConnectionString="..." />

</Globals>

Where the ‘DataConnectionString’ is set to any valid SQL Server connection string.

In order to use SQL Server as the store for system data, there’s now a script file ‘CreateTables.sql’ in the ‘SDK\Binaries\OrleansServer’ folder which establishes the necessary tables with the right schema.

Make sure that all servers that will be hosting Orleans silos can reach the database and has access rights to it! We’ve tripped up a few times on this seemingly trivial concern during our own testing.