Using Omniscient Foundation

# Application Manager

The application manager is the entry point to the foundation. It serves as a bootstrapper for your application; by calling the StartApplication method, you will have all your services loaded and started. During this invocation, the application manager will also instantiate default objects if you didn’t provide them *before* calling StartApplication. For example, if you wish to override the default PresentationController, you may set a value to ApplicationManager.Current.PresentationController before calling StartApplication. The application will then use your own presentation controller instead of the default one.

# Services

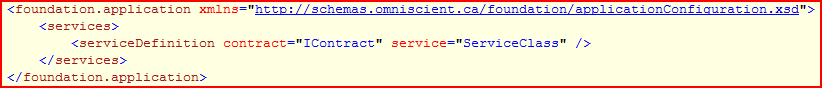
Services are special “modules” that implement a public contract. The contract is defined by an interface. The service is therefore a module that implements an interface, basically. Services have a name, and are able to return an instance of an object that implements the contract. Separating the service and the contract implementation helps decouple de code even more.

## Lifetime

When the application starts, it loads all services. That is, an instance of the service is created and kept in a container. That means the service itself is a singleton; it won’t be instantiated twice. However, since the service is responsible for returning an instance of an object that implements the contract, it is free of using either singleton or single-call approaches into managing the lifetime of that implementation object. If the implementation object benefits from being managed as a singleton, then the service may create the instance at construction time, and always return the same instance. On the other hand, if the implementation object benefits from being single-call, then the service may just return a new instance each time it is asked for the implementation.

## Creation

Services may be defined in the application file, or manually added just before calling StartApplication. Here’s how to define a service in the configuration file:

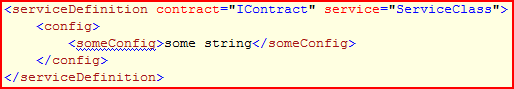


*Contract* and *service* attributes must lead to an existing type, or the form “namespace.class, assembly”. It’s pretty much the same idea when you manually add services to the ApplicationManager : you call ApplicationManager.Current.ServiceContainer.RegisterService, passing the type of a contract and the instance of a class implementing IService.

To easily create a service, you may derive from two existing classes: GenericSingleCallService and GenericSingletonService. As their name imply, one has a single-call way of managing the implementation object, while the other treats the implementation object as a singleton. You may also prefer to implement IService<TContract> directly in your service class.

## Configurable Services

Sometimes a service may wish to receive configuration upon being loaded. That is automatically supported by the ApplicationManager, as long as your service class implements ServiceModel.IConfigurable. All you have to do is define a configuration in the configuration file as follow, and the Configure method of your service will be called when the application starts, passing it your xml configuration element :



In this case, <someConfig> will be passed to the Configure method of your service. Note that you may not use a sequence under the <config> node.

## Startable Services

Some services may support the concept of being *started* and *stopped*. In that case, those services will be invoked during ApplicationManager’s StartApplication and CloseApplication invocations. Every service that implements ServiceModel.IStartable (even those added manually to the application) will be invoked during application’s start and stop phases.

# Object Container

The object container is accessible through the ObjectContainer property of ApplicationManager. It has a default implementation – assigned during the call to Start Application – but it may point to any class that implements IObjectContainer.

The object container is really simple; it allows you to register an object, and latter get it back by using its type. Only one object of a given type may be registered at the same time; it is an error to try and register two different objects of the same type.

# Entities

Entities are objects that represent pure data. An Entity is the object representation of a row in a table. Every entities have an Id that correspond to a surrogate key (defined as a GUID) that all tables must have in the database. This is a limitation we are aware about, and we believe that the benefits outdo the constraints of adding a surrogate key to all tables of the database. In particular, this allows huge benefits for the MVC (read below about the MVC pattern).

## Entity Status

Every entity has a status to monitor its lifetime in regard to the database. The statuses are listed here:

* **NotLoadedYet**: The Entity has been instantiated from the user code and is known to be existent in the database, but hasn’t been loaded yet. It is possible to set the Id of such an Entity.
* **New**: The Entity has been instantiated from the user code and is known to be inexistent in the database. Saving such entity will lead to an INSERT statement.
* **Clean**: The Entity has been instantiated by an Adapter, and it exists in the database. Its properties’ values reflect those of the data row in the table.
* **Dirty**: The Entity has been first instantiated by an Adapter with a Clean status, and has been modified since then. Saving such entity will lead to an UPDATE statement.
* **ToBeDeleted**: The entity has been instantiated by an Adapter with a Clean status, and has been marked after that to be deleted. Saving such entity will lead to a DELETE statement.
* **NonExistent**: The entity was either ToBeDeleted and has been saved (deleted from the database), or it was New and been marked as deleted. Such entity is useless, and unable to be saved.
* **Clone**: The entity is the result of a Clone operation made on another entity. Such entity is unable to be saved.

Most of the time, you won’t have to manually change the status of an entity. It’s the job of entity controllers to do that.

## Entity Controller

The entity controller manages the status of entities. It also automatically keeps track of the “original” data of an entity, when you wish to safely edit the entity. It is also responsible for loading and saving entities, with the help of entity adapter (covered shortly).

# Presentation and the MVC Pattern

An implementation of the MVC pattern has been developed into Foundation. It is made of a Model, a View and a ViewController. Basically, its role is to display entities on the UI. You wrap an Entity into a Model, and you ask the ApplicationManager “Would you please display that Model?” ; ApplicationManager then asks all registered ViewControllers to “open a view” for that Model (generally, you will have very few view controllers – often only one. And generally, only one of them will open a view for that model).

Here’s the basic idea: a Model wraps data; it is loaded anywhere in the application, and doesn’t know how to “paint” itself on the screen. A view is able to display and edit a model. The view is unable to place itself into a window, and therefore is unable to really display itself. That’s the job of the ViewController: it knows about the application window and display regions, and it knows how to display views (views are generally user controls). The view controller generally manages a region of the main window. Sometimes, it may also have nothing to do with the main window, and instead be able to open popup windows, for example.