Contents

[Chapter 3. How Windows Azure works 1](#_Toc298777463)

[Windows Azure, an operating system for the cloud 1](#_Toc298777464)

[Managing the assets of the cloud 2](#_Toc298777465)

[The Fabric Controller 2](#_Toc298777466)

[The Fabric Controller is just an application 2](#_Toc298777467)

[How the Fabric Controller works: driver model 3](#_Toc298777468)

[Allocating Resources 3](#_Toc298777469)

[Instance Management 4](#_Toc298777470)

[The service model and you 4](#_Toc298777471)

# Chapter 3. How Windows Azure works

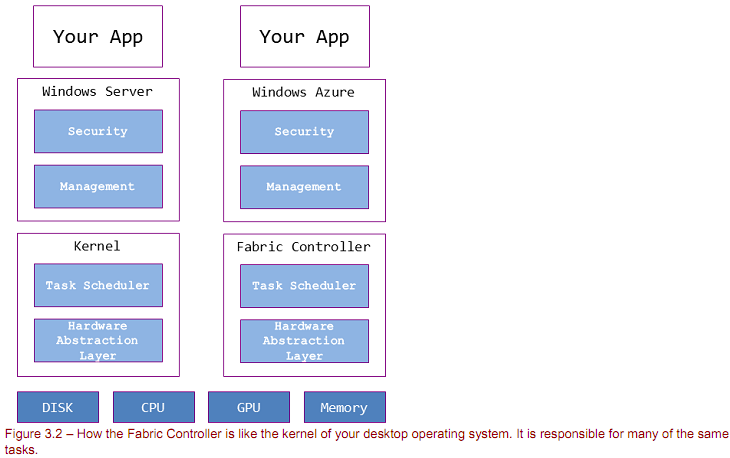
This chapter covers

* How Microsoft built Azure
* What a cloud operating system is
* How your application is provisioned and managed in the cloud

## Windows Azure, an operating system for the cloud

Windows Azure, the cloud operating system, does the same work as the desktop operating system, but on a grander scale, and for distributed applications.

The fabric is comprised of the thousands of servers running, working together as a cohesive unit.



Windows Azure takes care of the whole platform so you can focus on your application. Understanding the relationships between your code, Azure, and the fabric controller will help you get the most out of the platform.

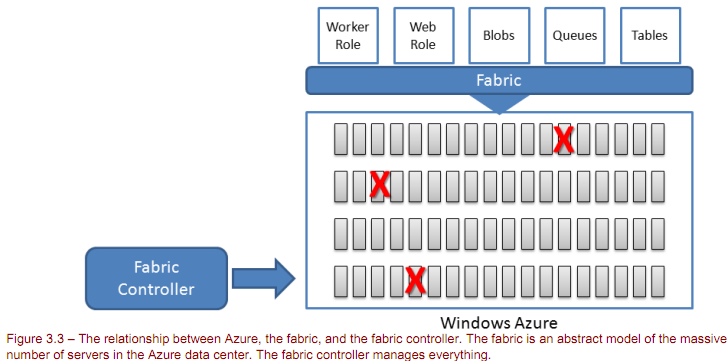
### Managing the assets of the cloud

In fact, Azure manages much more than just servers. There are plenty of other assets that are managed. In addition to servers, Azure manages routers, switches, IP addresses, DNS servers, load balancers, and dynamic VLANs.

If the Azure fabric is the operating system, then the fabric controller is the kernel.

## The Fabric Controller

Operating systems have, at their core, a kernel. This kernel is responsible for being the traffic cop in the system. It manages the sharing of resources, schedules the use of precious assets (CPU time), allocates work streams as appropriate.



The Azure fabric has a kernel called the *fabric controller*, or FC. It handles all of the jobs a normal operating system’s kernel would handle. It manages the running servers, deploys code.

### The Fabric Controller is just an application

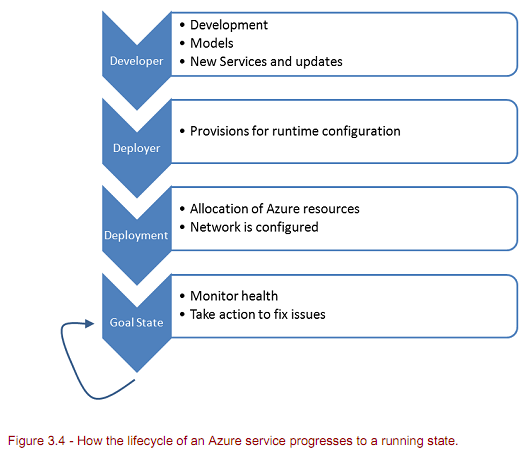
The FC is an Azure application in and of itself, running multiple copies of itself for redundancy sake.

The FC contains the complete state of the fabric internally, and this is real time replicated to all of the nodes that are part of the FC. In case one of the primary nodes goes offline, the latest state information is available to the remaining nodes, which then elect a new primary node.

The FC manages a state machine for each service deployed, setting a goal state based on what the service model for the service requires. Everything the FC does is in an effort to reach this state, and then to maintain that state once it is reached.

### How the Fabric Controller works: driver model

The FC follows a driver model, just like a normal operating system. The FC works with a series of drivers, for each type of asset in the Azure fabric. This includes the machines, as well as the routers, switches, and load balancers.



The FC uses these drivers to communicate the commands it needs to send to each device to reach the desired running state. This might be the command to create a new VLAN to a switch, or allocate a pool of virtual IP addresses.

Figure 3.4 shows the progression of state, from the developer writing the code, and defining the service model, to the FC allocating and managing the resources the service requires.

### Allocating Resources

One of the key jobs of the FC is to allocate resources to services. It does this by analyzing the service model of the service, including the fault and update domains, and the availability of resources in the fabric. It finds which nodes can support the needs of each instance in the model.

Once it has reserved the capacity, it updates the FC data structures in one transaction.

### Instance Management

The FC is also responsible for managing the health of all of the nodes in the fabric, as well as the health of the services running.

If it detects a fault in a service, it will try to remediate that fault, perhaps by restarting the node, or taking it offline and replacing it with a different node in the fabric.

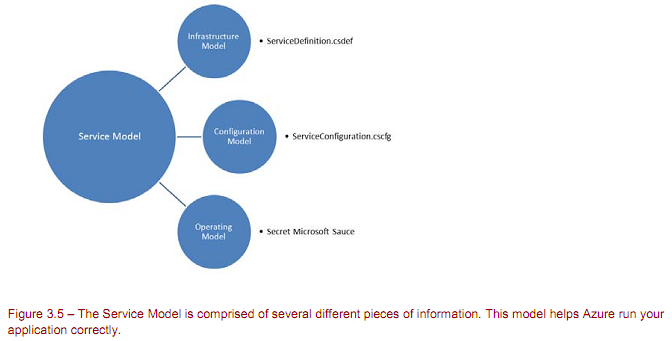
When a new container is added to the data center, the FC performs a series of burn-in tests to ensure that the hardware delivered is working properly, maks it available to be allocated by the FC.

If hardware is ever to be determined to be faulty, either during installation, or during a fault, the hardware is flagged as unusable in the inventory, and left alone until later. Once the problems have been fixed, the whole container is retested and returned into service.

## The service model and you

The driving force behind what the FC does is the service model that you define for your service. You define the service model in an indirect manner. When you are developing a service, you define the following:

* some configuration on what the pieces to your service are
* how the pieces communicate
* expectations you have about the availability of the service



The service model is broken into two pieces of configuration and deployed with your service.

### Defining configuration

Your solution in Visual Studio will contain these two pieces of configuration in different files, both found in the Azure Service project in your solution:

* service definition file (*ServiceDefinition.csdef*)
* service configuration file (*ServiceConfiguration.cscfg*).

The service definition file defines what the roles are in your service, and what their communication endpoints are. This would include public HTTP traffic for a website, or the endpoint details for a web service. The service definition cannot be changed at runtime. You can think of this piece of the configuration as defining what the infrastructure of your service is, and how the parts fit together.

The service configuration file includes the entire configuration needed for the role instances in your service. Each role has its own dedicated part of the configuration. The contents of the configuration file can be changed at runtime.

You can also access the configuration in code, in a similar manner that you might read a *web.config file* in an ASP.NET application.

### Adding a custom configuration element

In many applications we store connections strings, default settings in the app.config or *web.config*. You will often do the same with an Azure application.

We first need to declare the format of the new configuration setting in the *.csdef* file. We do this by adding a ConfigurationSettings node inside the role we want the configuration to belong to. We do this to define the schema of the *.cscfg* file for that role.

<ConfigurationSettings>

<Setting name="SuperSecretPassword"/>

</ConfigurationSettings>

We add the following XML into the appropriate role node in the *.cscfg* file.

<ConfigurationSettings>

<Setting name="SuperSecretPassword" value="KlatuBaradaNikto"/>

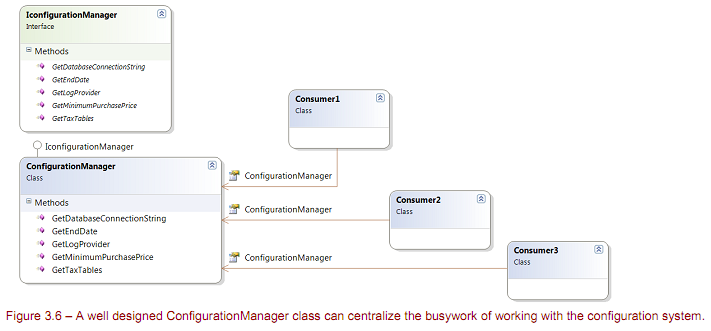
</ConfigurationSettings>

During runtime, we want to read in this configuration data, and use it for some purpose. Remember that all configuration settings are stored as strings and must be casted to the appropriate type as needed.

txtPassword.Text = RoleEnvironment.GetConfigurationSettingValue("BannerText");

### Centralizing file-reading code

It is a best practice to take your entire configuration file reading code, where ever it is sprinkled, and move it into a ConfigurationManager class of your own design. This centralizes all of the code that knows how to read the configuration in one place, making it easier to maintain.

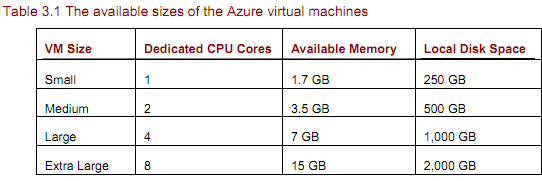


While keeping them in place reduces the amount of change to your code as you migrate it to Azure, it does come at a cost. Unfortunately the configuration files that are part of your roles are frozen, and are read only at run time. You cannot make changes to them once your package is deployed. If you want to change settings at runtime you will need to store those settings in the *.cscfg* file. Then when you want to make a change, you only have to upload a new *.cscfg* file, or click configure on the service management page in the portal.

### The many sizes of roles

Each role defined in your service model is basically a template for a server you want deployed in the fabric. Each role can have a different job, and a different configuration. Part of that configuration includes local storage, and the number of instances of that role that should be deployed. How these roles connect and work together is part of why the service model exists.

Since each role might have different needs there are a variety of virtual machine sizes that you can request in your model.



Which size you want is defined in the *ServiceDefinition.csdef* file, on a role by role basis. The default size, if you don’t declare one, is small.

<WorkerRole name="ImageCompresser" vmsize="ExtraLarge">

If you are using Visual Studio 2010, you can define the role configuration in the properties screen of the role in the Azure project:

