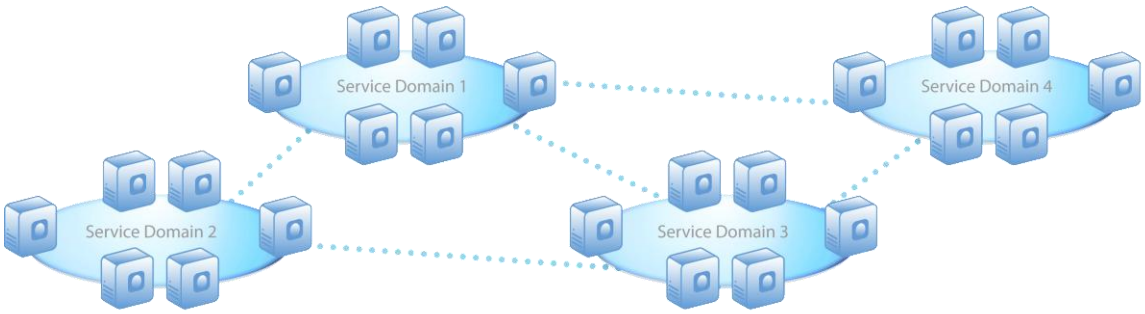
Configuration Service 5.0 Technical Guide

6/9/2011



Microsoft .NET Windows Azure Microsoft SQL Azure

THIS IS NOT A PRODUCT SPECIFICATION.

This document and related sample code supports Windows Server® 2008 R2, Windows Azure, SQL Azure, Azure AppFabric and the Microsoft .NET Framework 4.0 as a redistributable sample application kit.

The information contained in this document represents the current view of Microsoft Corp. on the issues disclosed as of the date of publication. Because Microsoft must respond to changing market conditions, this document should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented. This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Microsoft grants you the right to reproduce this guide, in whole or in part.

Microsoft may have patents, patent applications, trademarks, copyrights or other intellectual property rights covering subject matter in this document, except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights or other intellectual property.

Microsoft, Windows Server, Windows Azure, SQL Azure, SQL Server, the Windows logo, Windows, Active Directory, Windows Vista, Visual Studio, Internet Explorer, Windows Server System, Windows NT, Windows Mobile, Windows Media, Win32, WinFX, Windows PowerShell, Hyper-V, and MSDN are trademarks of the Microsoft group of companies.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

Introduction 4

Core Benefits of Configuration Service 4

What is Configuration Data? 5

What the Configuration Service Provides 5

Implementing the Configuration Service 6

Windows Communication Foundation Core Concepts 6

What is a WCF Service? 6

Core Components of a WCF Service 7

Configuration Service Overview 8

Load Balancing and Failover 11

Implementing the Configuration Service 12

Technical Documentation 15

Proper DNS/Network Setup 15

Firewalls 15

Debugging Your Host Using Visual Studio 15

Adding Custom Settings to Be Managed in Repository 15

Adding Validation Logic to Custom Configuration Settings 17

Storing Custom Classes/Types in the Repository 18

Executing Logic on Configuration Changes to Custom Settings 19

Securing Configuration Service Endpoints 19

Navigating ConfigWeb 20

Using the Provided CustomUserNameValidator and CustomCertificateValidator Classes 20

Adding Host-Side and Client-Side Binding Configurations 21

Always Define Client Configuration Sections and Client-Side Bindings In Your Host's Application's Configuration File for Hosted Service Endpoints 22

Modifying Node and Configuration Service Template Bindings 22

Securing Configuration Service Endpoints 22

Adding Client Configuration Sections 23

Generating Client Binding Configurations and Client Configurations with SVCUTIL.EXE 23

Adding Service Behavior Configurations 24
Adding Endpoint Behaviors on the Host-Side 24
Adding Endpoint Identities on the Host-Side 24
Adding Endpoint Behaviors on the Client-Side 24
Adding Endpoint Identities on the Client-Side 24
Naming Your Hosted Service Endpoints 24
Creating Virtual Hosts and Service Endpoints for IIS-Hosted Services 25
Establishing Connected Service Definitions to Remote Services 25
Connecting to Primary Remote Services via ConfigWeb 26
Connecting to Generic Remote Services via ConfigWeb 26
Changing Hosted Service Definitions Once Clients are Already Subscribed to an Endpoint 26
Dynamic Clustering for Load Balancing and Failover 27
Removing Downed Nodes Manually 28
Removing Clients Manually 28
Load-Balancing Against Java Application Server Services or other Non-.NET/Non-Windows based Hosts 28
Polling Logic 29
Creating Service Endpoints that Listen on an Explicit IP Address or DNS Name 29
Creating Multiple Virtual Hosts within One Master Process 30
Creating Services that Implement Multiple Contracts 30
Using the Repository Creation Tool 30
Generating and Using the Client Class 30
Important Callbacks on ChannelFactory Creation 31
Important Calls to Create and Manage Your Own ChannelFactories and Channels 32
Important Callbacks after ServiceHost and WorkflowServiceHost Creation 33
Working with Duplex Contracts 33
Management of Lifecycle for IIS 7.0 WAS-Hosted Services 35
CustomActions 35
Can Nodes be Geographically Dispersed? 36
Is the Configuration Repository a Single Point of Failure? 36
Node Notification Failures 36
Using the Configuration Service Programmatically For Automated Actions 36

ConfigWeb and Service Domain Location Independence 37
The Best Physical Network Setup for Deployed Scenarios 37

Introduction

This document provides an overview of the Configuration Service implemented in the .NET StockTrader 5.0 sample application as posted on MSDN (http://msdn.microsoft.com/stocktrader). The Configuration Service is an independent service, and can be implemented in any application or service via the shared libraries and the base class implementation that have been published at this site. Configuration Service 5.0 introduces a Visual Studio 2010 template and wizard that enable a base implementation to be built without writing any code. This template produces a well-structured Visual Studio solution based on the StockTrader 5.0 design pattern (including a sample data access layer); and this solution also includes the two projects and logic for a complete implementation of Configuration Service 5.0. Simply use the template, pick your hosting environment, start with the base solution and add your own business logic and data access logic. All of the features (including ConfigWeb) are then available for scaling your application or service across multiple nodes. All host types are supported: IIS Web application, .NET Console Application, .NET Windows NT Service, .NET Windows application, Windows Azure Web Role, Windows Azure Worker Role.

Core Benefits of Configuration Service

The following are the core benefits/features of the Configuration Service:

- **Centralized storage** and management of configuration data for .NET-based applications and service hosts, whether on-premise or Azure-deployed. Configuration data is stored in a database, but treated just as if it came from an <appSettings> section of a .NET configuration file. Settings are local static variables, cached in-memory for high performance. However, they are kept in sync across scale-out service nodes.
- **Linked services** with secure configuration data gateway: The ability to view and reconfigure connected services n-levels deep within a service-oriented composite application, even for services connected via physically separated subnets (for example, connected services might be deployed in different data centers or application hosting environments and geographic locations).
- **Live updates** to service configurations without redeploying configuration files or service host/application restarts.
- **Dynamic clustering** of on-premise service nodes, without any manual operations required to initially set up and maintain a running cluster of service nodes.
- **Automatic synchronization** of configuration updates across all clustered service nodes.
- **Load-balancing** of service operations (remote service requests) across multiple servers for horizontal scaling of middle-tier services.
- **Application-level failover** of service nodes running a service, across clustered nodes.
- A central, common **Web-based management application** (ConfigWeb) for service-oriented applications.
- **Basic health monitoring** of service nodes in a cluster.
- The ability to **support multiple types of bindings**, including transport protocols, security/network settings and message-encoding standards, between service clients and service hosts, simultaneously.

What is Configuration Data?

Configuration data can be any information or metadata that defines how a service will operate. For example, in the .NET StockTrader Web application, a configuration setting determines how many orders to display on the Account Summary page. Another configuration setting determines whether to run the Business Service layer remotely, or in-process on the same computer as the Web application. Yet other configuration settings define which transport protocol (http/text-XML, tcp/binary encoding, etc.) to use to connect to the business service layer; or which services to expose on specific endpoints when constructing the .NET runtime. The .NET StockTrader business service layer also has many of its own configuration settings, including what database to connect to (including whether SQL Server 2008, SQL Azure or Oracle 10/11g), and the location of the StockTrader production database. Configuration settings also determine how to manage transactions, transaction timeouts, etc.

Any custom application-specific configuration information that might otherwise be embedded in the **<appSettings>** section of a .NET configuration file, or hard-coded, can be setup and managed via the Configuration Management Service simply by defining new configuration keys to store the information. Configuration keys are automatically loaded into the service host at startup, and changes can be made dynamically after startup via the Configuration Management user interface, ConfigWeb. Such changes can be made without deploying new configuration files, which would also require application restarts on running nodes.

Any type of setting a developer wants to define can be managed by the system. It is up to the developer of the system to actually utilize the settings in their application as part of their business logic. However, several common settings are also defined and managed automatically by all services/application hosts implementing the Configuration Service. These are application-server specific settings the Configuration Service automatically provides (and can be extended). What additional application-specific settings might you want to store? Anything you want to be configurable dynamically after deployment, without recompiling and re-deploying code or configuration files, and restarting host processes! For example, even the default sizes for textboxes in your Web or Windows application might be defined as a configurable setting, which an administrator/operator could change at a later time via ConfigWeb. Also, keep in mind these updates are applied live across all running clustered nodes; and any new node added later will initialize with the same setting value since it is stored in the repository.

What the Configuration Service Provides

The Configuration Service is meant to make it easier to dynamically manage the configurations and ongoing operations of distributed, service-oriented applications. The Configuration Service also provides the ability to automatically cluster services on multiple servers (service *nodes*), and load-balance across these nodes from service clients with application-level failover. If a service node goes down, clients will automatically detect such failures (whether a network, physical server or application/service host software failure), and redirect requests to other online nodes. In this fashion, a service host is virtualized across servers without the need for external hardware or software-based load balancers, although the Configuration Service can also integrate with such external load-balancers.

Once a service is virtualized in such a fashion, the Configuration Service Management Web (ConfigWeb) allows administrators to make configuration updates to a service as well as any connected services, and automatically update all running service nodes without the need for application re-starts or manual configuration synchronizations. Hence, the Configuration Service is meant to make service-oriented applications more robust, reliable, and easier to manage. All of this is accomplished via the use of SQL Server-based ***service configuration repositories***. The Configuration Service itself is implemented using .NET 3.5 and Windows Communication Foundation (WCF), and is itself based on industry Web Service standards.

Implementing the Configuration Service

The Configuration Service is contained in pre-compiled .NET assemblies, and while implemented by the .NET StockTrader application as an illustration, is completely independent of the .NET StockTrader application itself. Hence, the Configuration Service is designed to be easily implemented by customers in any custom application or business service. Simply use the new Visual Studio 2010 template provided—no code is even required to have a working application within minutes. In addition, any base class method can be overridden, and custom functionality provided for extensibility purposes. The ConfigurationActions class can also be extended to capture changes to specific configuration settings and execute custom logic on every clustered node. Once implemented, the management user interface, ConfigWeb, can attach to a custom application or service host via the login page, and ConfigWeb can be fully utilized to manage custom applications, services and complete composite/distributed applications. The 5.0 version also includes a tool that makes it easy to create an initial service configuration repository for a custom service (as illustrated in the tutorial).

Windows Communication Foundation Core Concepts

Windows Communication Foundation is Microsoft's strategic; platform-level service oriented programming model and runtime for .NET. For those new to Windows Communication Foundation (WCF), this section introduces some core WCF concepts and terminology that are necessary to understand before reading further. Readers should also visit the WCF site on MSDN for both conceptual overview information about WCF, and detailed technical documentation.

What is a WCF Service?

A WCF service is a black-boxed set of functionality, implemented in the .NET Framework, which provides functionality to clients that can remotely invoke the service operations over industry standard protocols and encoding standards. WCF is based on Web-service standards such as HTTP, SOAP, WS-* standards and XML, but also supports other de facto standards including REST/POX, MTOM, Tcp, .NET binary encoding, MSMQ message queuing, .NET distributed transactions, and many other standards. Java-based and .NET-based clients can easily connect to WCF services, and WCF clients can easily connect to both .NET and Java-based services, as illustrated in the .NET StockTrader application. If you are new to WCF, it is highly recommended you visit MSDN and take one or more of the online tutorials or install/examine some of the basic code samples available. WCF is a .NET 4.0 component, so you will

want to make sure you have the .NET 4.0 runtime and the latest Microsoft Window SDK installed first (both are free downloads on MSDN).

Core Components of a WCF Service

A WCF service consists of (at least) the following core components, typically isolated within different Visual Studio projects/assemblies for clean partitioning:

1. The **Service Contract**. This is typically an interface that simply defines the methods (service operations) that the service will support/implement. The service contract is attributed with WCF attributes, much like ASMX-based Web Services were attributed with the [Web Method] attribute.
2. The Service **Data Contract.** The data contract simply defines any custom classes/types that will be passed between clients and services, either as parameters or return values. These will automatically be serialized by WCF for over-the-network transport, typically (but not limited to) text-XML/SOAP encoding or .NET binary encoding.
3. The **Service Implementation**. The service implementation is the class that actually implements the service functionality, and inherits from the service contract, providing the implementation logic for all the service operations (methods) defined in the service contract. Service operations might be self-contained (for example, communicating with a database); or may invoke other remote services or any .NET component/logic. So a service might also be a client to other services with service-to-service interactions across the intranet, an extranet, or the public Internet.
4. Service **Bindings**. Bindings are the network transports and encoding standards used by a service. A WCF service can simultaneously support multiple bindings—listening on both HTTP and TCP endpoints, for example, potentially using different security and message-encoding standards on different endpoints. Clients can choose which binding to use to connect to a WCF service, based on the endpoint they are connecting to. With WCF, binding information can be stored in application configuration files or constructed programmatically. Typically, once tuned, binding information does not need to change. The Configuration Service uses WCF Binding Configurations, with the configuration name(s) stored in the repository, pointing to specific binding information defined and loaded from the application-specific configuration files (web.config or .exe.config). Many template bindings are provided with the Configuration Service 5.0.
5. **Service Host**. A service host is a program (process) that hosts one or more services. It is a physical executable process. With WCF, the most common service host is Internet Information Server (IIS), which runs the service(s) under the IIS administration and security infrastructure within an ASP.NET worker process (w3p.exe). However, services can also easily be self-hosted in custom executables, such as .NET Windows applications, console applications, or .NET-based Windows NT Services.

The Configuration Service itself is a WCF service. It can be hosted in IIS, or in custom self-host programs. The Configuration Service also has a sister service, the Node Service, that handles communication between peer nodes in a cluster setup. When implementing the Configuration Service (either in IIS or a

self-host program), the service host will actually be hosting custom service(s) you define, as well as the Configuration Service and the Node Service. Developers can customize the bindings (transport protocol, encoding standard, and security settings) for their host-specific Node and Configuration Service endpoints.

Configuration Service Overview

A service-oriented application makes it possible to construct *composite applications*---applications consisting of multiple services interconnected to provide the business functionality. With a composite application, such as the .NET StockTrader Web application, much if not all of the business functionality may not reside with the user-activated application itself, but rather it will likely be provided by remote services running on distributed servers, possibly on the other side of a firewall, or even in remote data centers or across the Internet. In addition, these services may be providing their functionality to multiple applications—hence being reused to support different business scenarios. Also, the services themselves may connect to other services, including services hosted by business partners or cloud-based service providers. A good analogy is a tree structure (as is used with Windows file/folder views). One folder might contain one or more other folders, each then containing their own folders. Hence, a network node map for a composite application is really a tree of connected services, and this is how the Configuration Service views a composite application. For example, consider the following illustration:

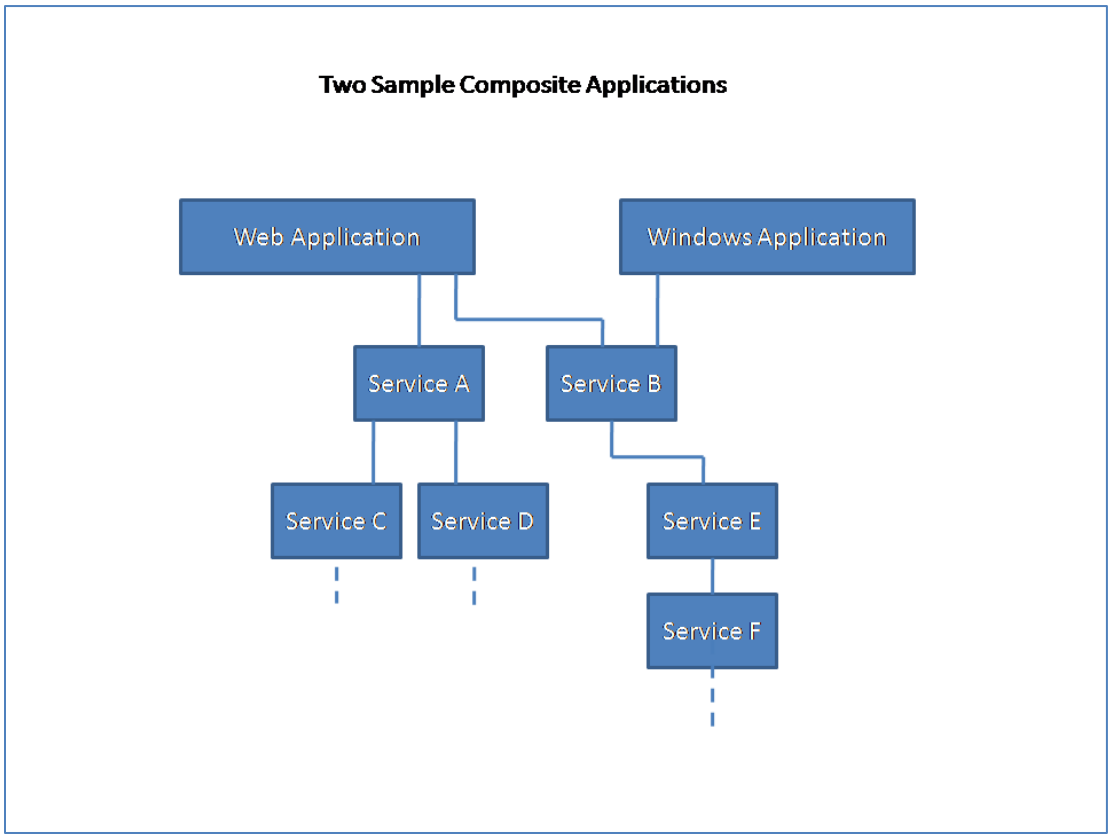


Figure 1: Sample SOA Topology

Here we can see two different root “composite” applications (one a Web application, the other a Windows application), each connecting to remote services that provide some or all of their business functionality. We can also see the tree-like nature of these connections, with each service possibly connecting to other services, n-levels deep.

The Configuration Service, once implemented, provides the ability, if the services are so configured and with a proper global administrator authentication, to allow the administrator to iteratively “drill down” into configuration information for connected services that also implement the Configuration Service, even through n-different levels of services even if the root service is not located on the same physical subnet as services several levels down. Also, the Configuration Service and ConfigWeb support the concept of connecting to multiple services, and to multiple service hosts, even to multiple service hosts each hosting multiple services. Again, just think of the tree structure analogy. *Connection Points* represent a connection from one application or service, to another remote service. So a Web application (or Windows application) may have multiple connection points to different services, which in turn might have their own connection points to other remote services.

One important concept of service-orientation that is followed in the Configuration Service is the notion that services should be **autonomous**. So in the illustration above, the Web application is completely unaware of services C, D E and F. Service A is aware of (connects to and understands the service and data contracts) Services C and D; while Service B is aware of Service E, but only E is aware of Service F. The Web application is aware of both A and Service B, since it connects directly to them. So services are autonomous, and clients that connect only need to understand the service contract, data contract (the service ‘schema’), and an endpoint (URI) to connect to in order to utilize that specific service. Hence, only schema information is shared, and implementation logic and service management is completely black-boxed.

This also applies to location independence. In the diagram above, services that directly connect must have network connectivity, of course. However, the Web application may reside on a completely different network (and geographic location) than services C, D, E and F—which may each reside anywhere. In many cases, a service-oriented deployment may involve connections only to services that reside within the same data center/network, but this is not assumed. Also, since WCF is based on Web Services, applications and services may be based on Java or .NET, and still interconnect. However, while the Configuration Service can help manage connections to non-.NET services and .NET services that do not also implement the Configuration Service, the Configuration Service will only be able to drill down and view/modify configuration data for services that also implement the Configuration Service. ConfigWeb can, however, be used to establish and manage connections to non-.NET services, and even provide load balancing/failover against non-.NET services and online/offline status information for the remote endpoints on non-Microsoft platforms.

So now consider the same illustration, however, add the notion of clustered service nodes:

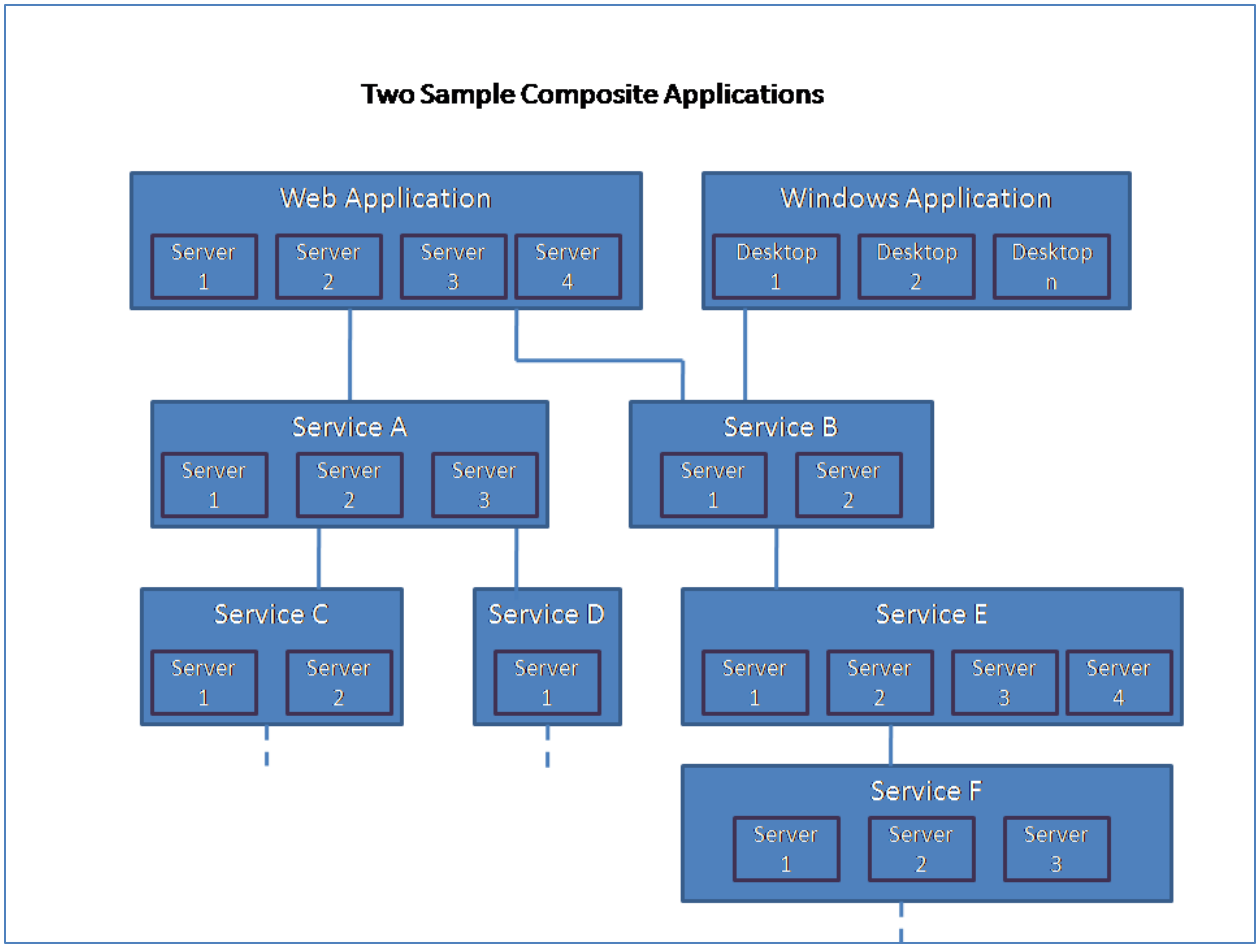


Figure 2: SOA topology showing services hosted on scaled-out clusters of servers.

As you can imagine, even just managing configuration data for a single service or application on such a cluster could be a challenge.

For the Configuration Service, the notion of a central store of configuration data is utilized, which we call a service configuration database (aka repository). SQL Server is used to store the configuration data. Also, to maintain the notion of autonomous services, each service host gets its own repository. There is no single repository; rather there is a shared schema for all repositories, and a shared way to access any repository via the Configuration Service and the ConfigWeb UI given a common administrator authentication. Without such a common administrator authentication, services can still connect, but viewing and changing configuration data will be limited within the composite application to a single service domain.

Each separate service hosted by a single service host (the illustration depicts each service host hosting only a single service) can have its service-specific configuration data stored in a repository shared only by peer clustered nodes within that service domain. Also, since the Configuration Service can drill down n-layers deep in a topology, ConfigWeb is able to provide a single UI and login to manage a complex service-oriented topology, all in one place.

Load Balancing and Failover

Peer service hosts are automatically clustered simply by sharing a single service repository. In other words, the service host (IIS application; self-host executable or NT Service) can be executed/installed on n-number of servers, and will automatically join the cluster simply by nature of the fact it points to a common repository database. Also, a "root" composite application can be clustered (for example, with hardware load balancing or software load balancing such as Windows Network Load Balancing), yet share a common repository such that all nodes load common configuration data and updates are automatically synchronized across application nodes in the cluster. A load-balancing client base class is provided such that service clients will automatically round-robin requests across online nodes available, and failover should one or more nodes become unavailable. A **notification system** built into the system allows new nodes to be started on new servers and automatically start receiving requests from any online clients it is supporting. This notification system is dynamic in that new clients and/or new service hosts can be started and stopped at any time, and the proper connections will automatically be setup and utilized by the system for any service implementing the Configuration Service. To implement the load balancing, you will simply inherit from a base load balancing client class in your service-specific WCF client, and establish an initial connection to a service from a client via ConfigWeb. If that service is virtualized/horizontally scaled across n-servers, the client application/service (also possibly clustered), will actually establish and use connections to all service host nodes available automatically. ConfigWeb can be used to view the physical nodes a service host is clustered across, and the online/offline status of that physical node. Separate physical clusters of the same service implementation could be established simply by having two distinct configuration repositories, with nodes in Cluster A connecting to service repository A, and nodes in cluster B connecting to service repository B. Each might be serving unique client applications, perhaps with different service level agreements (SLAs).

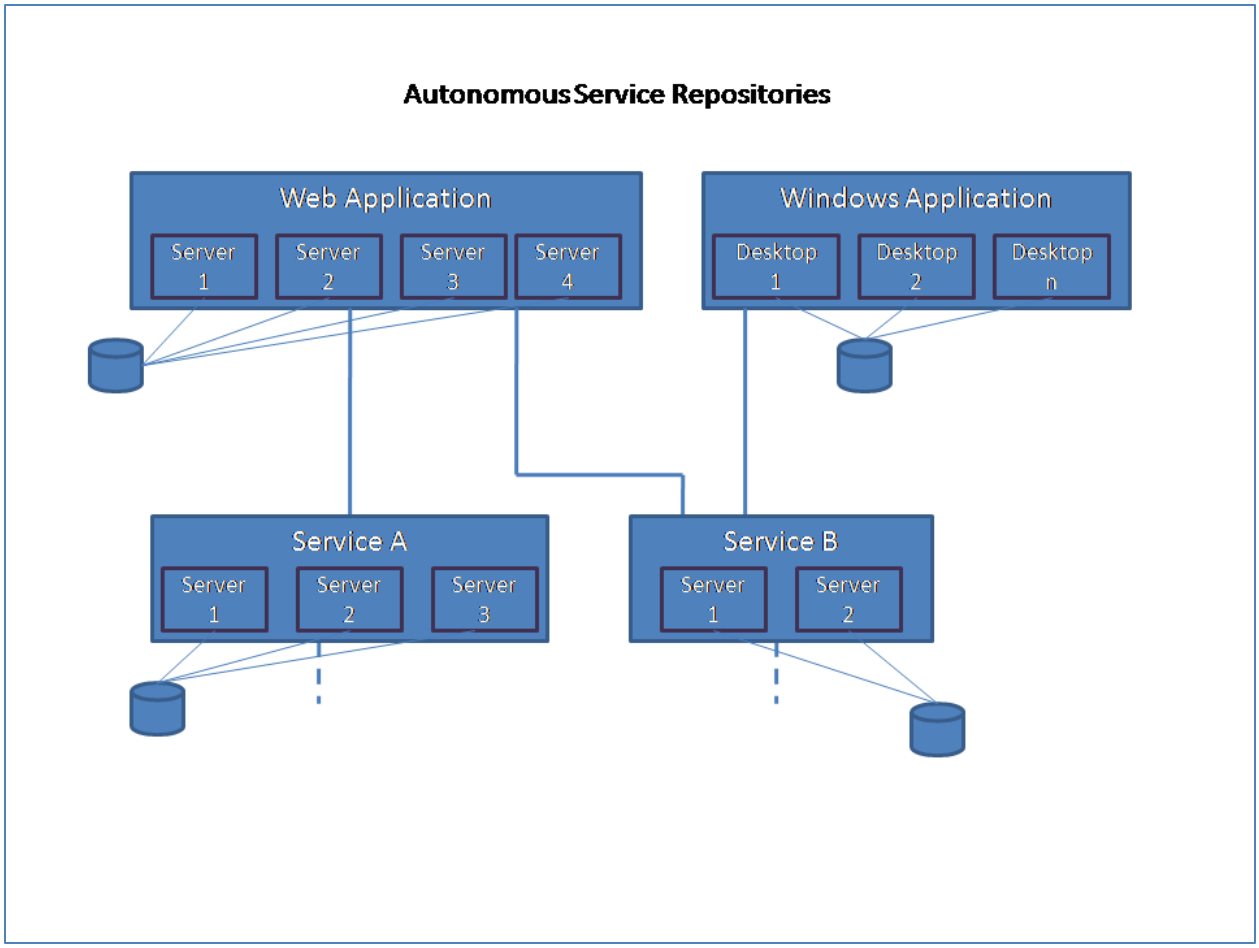


Figure 3: Autonomous Service Configuration Repositories per Service Host. Each service repository is shared by peer nodes, but distinct per service host/application. Also, every service configuration repository shares a common schema.

Implementing the Configuration Service

In order to implement the configuration service, you will need to create the following elements (the separate tutorial in the separate document entitled *Guide to Implementing the Configuration Service 5.0 for .NET Applications and Services* explains how step-by-step, and should take less than an hour to complete).

1. **A service repository** which is a database with a common schema. This database is created and initially populated via a tool provided with the download. This will automatically create the tables and pre-populate configuration data for the Configuration and Node services.
2. **A Settings project and class**. This is simply a .NET class that will be setup in a new Visual Studio project, and inherit from a base class implementation already provided. The Settings class defines (but does not initialize) the fields (variables) that will contain configuration data for your application or service—anything you might otherwise be storing in the <appSettings> section of a web.config or exe.config file, for example. You can define any number of custom settings and create service repository "keys" to store the settings later. Initially, you do not need to create

any custom keys, just create an empty class that inherits from the SettingsBase class already provided in the shared libraries. The base class defines common host settings all applications and services that implement the Configuration Service will have. You can think of these as application-sever specific settings.

3. **A Configuration Service Implementation project**. This will be the project/namespace that contains your Configuration Service implementation class and Configuration Actions class. Hence, this project will have two simple classes defined in it (items 4 and 5 below).
4. A **ConfiguationService class**, defined within the service implementation project called out as item 3 above. This class will simply inherit from a base class. It is generated by the Repository Creation tool; then simply added to the Configuration Service Implementation project.
5. A **ConfigurationActions class**, defined within the service implementation project called out as item 3 above. This class will also inherit from a base class provided, and any custom logic provided later, if any is necessary. For many applications and services, just an empty definition inheriting from the base class will suffice and no additional logic beyond the base class logic already implemented will be needed. This will be generated by the Repository Creation tool; then simply added to the Configuration Service Implementation project.
6. If self-hosting, you will need to create a **self-host program**. This can be a Console Application, a Windows Application, or a Windows Service. The initial code will be generated by the Repository Creation tool. If hosting in IIS, your host application will be an ASP.NET Web application/project. A complete Windows host console is provided as a base class, and can be used as a convenient way to host services in a Windows application that provides monitoring functionality.
7. An **app.config** (compiled to .exe.config for executable apps) or **Web .config** file (for Web apps). The appropriate configuration file type will be generated by the Repository Creation tool.
8. If IIS-hosting, you will need to have a **.svc file** (single line of code) defined for your custom service(s), as well as the Configuration and Node services, and present within your Web-published virtual directory. Typically, the Configuration Service endpoint will be stored in **config.svc**, and the Node Service defined in **node.svc**. These will be generated by the Repository Creation Tool.
9. If you are defining a custom business service that will be hosted by your service host beyond just the Configuration and Node services, you will of course need to setup your WCF service contract, data contract and WCF business service implementation logic, typically done in separate projects under a common root namespace, ala the StockTrader elements. Hence, you would have a service contract project, service data contract project and service implementation project for each service you will be hosting, with this logic up to you to implement to support your business functionality.
10. If your application or service is connecting to any services, you will need to define a **custom WCF client** that will inherit from a base class load-balancing client provided with the Configuration Management System shared libraries. The Repository Creation Tool will generate this client automatically, if you provide a compiled service contract and data contract. The service contract and data contract might be generated via **svcutil.exe** (a tool provided as part of the Microsoft Windows SDK to aide in generating proxies to any industry-standards based service); or

provided directly as an assembly. In both cases, the Client Generation within the Repository Creation tool will generate a new client class from the existing information. The tutorial steps through this.

Technical Documentation

Proper DNS/Network Setup

The Configuration Service is sensitive to proper setup of the network and DNS/AD structure. It is imperative that the network be properly setup, with nodes able to communicate to each other over DNS names and/or IP addresses overridden in configuration files as discussed later in this document. If you are not on a domain with a DNS server; you can always use the \windows\system32\drivers\etc\hosts file to assign IP addresses to computer names in your cluster (for computers in a Workgroup, vs. a Domain, for example). A sure sign something is wrong is the SOA Map View; this will show across all nodes whether they can communicate properly with each other over the designated service endpoints. By default, the Configuration Service will use a DNS-provided name (typical AD domain setup) or the local computer name (Workgroup setup without AD or a DNS server) to determine the base address for service endpoints. However, this can be changed as well (to bind specific endpoints to alternate names or specific IP addresses, as discussed later in this guide).

Firewalls

Firewalls will likely be a common setup issue. You must enable firewalls to allow communication between nodes in a cluster. On Windows Vista, even the primary http port 80 is blocked by default, and must be enabled. Again, the SOA Map View will immediately show if all nodes can communicate, both inbound and outbound, with each other and to Configuration Service and Primary Service endpoints. Specific ports might need to be enabled, depending on your hosted service environment and the ports you choose to use.

Debugging Your Host Using Visual Studio

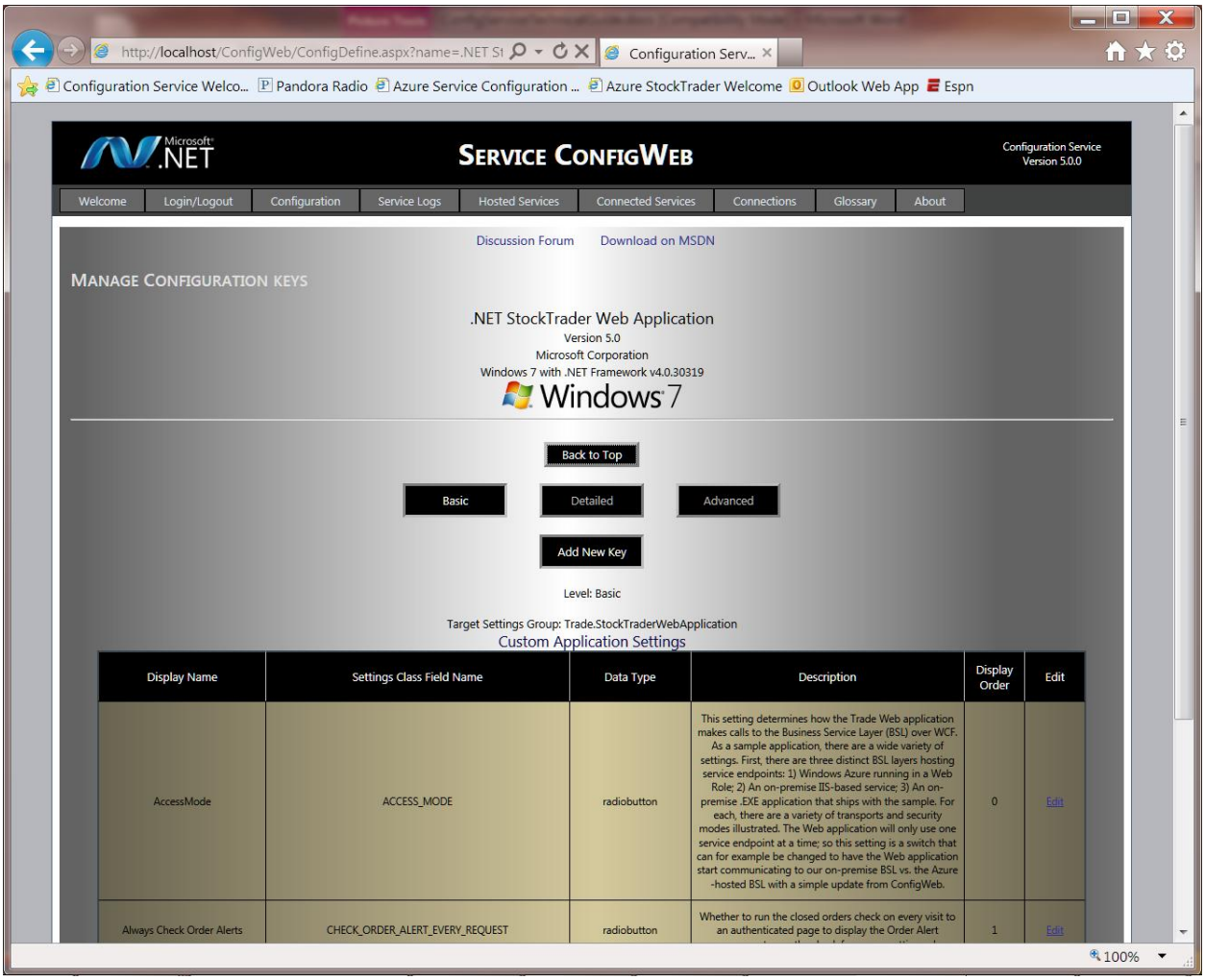
For IIS-hosted services, you should always make sure to use Visual Studio "Attach to Process" to debug, rather than hitting F5-go. You will need to create a real virtual directory in IIS for your Web application. Hitting F5 go on a Web site is not advised, because Visual Studio creates a separate IIS process/worker pool that is not linked into the actual IIS worker process; and hence other nodes (or ConfigWeb operations) communicating with your application are actually communicating with the IIS-hosted worker process, not the Visual Studio-created worker process. This is solved simply by attaching to the W3P.exe worker process via Visual Studio's Debug menu, and not using F5 Go to launch Web applications within Visual Studio itself.

Adding Custom Settings to Be Managed in Repository

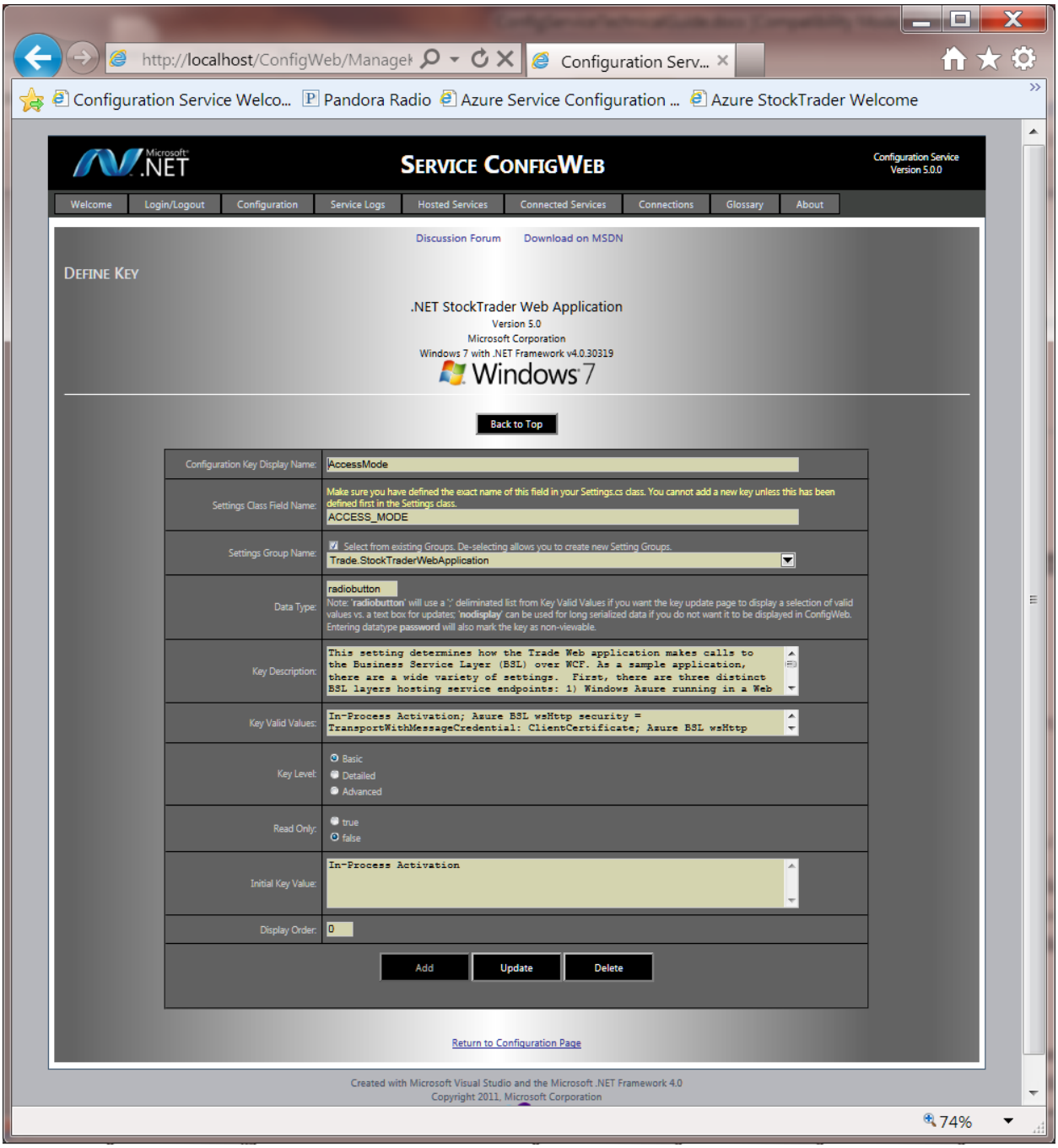
ConfigWeb is used to add custom settings to your application host's repository. You should consider putting into the repository most (if not all) settings you might otherwise manage in <appSettings> in configuration files (such as web.config); or as defined constants in your code. Once these become dynamically managed, you might find advantages to adding other changeable settings to your application. New with the 5.0 release is the ability to store XML-serialized custom classes in the repository, such that custom types/classes can be managed as with simpler settings types.

When you choose **Edit Configuration** for any host-level or sub-level settings group in the ConfigWeb home page, you will see a **Define/Edit Keys** link button. Depending on the settings group you chose to

edit, you will be presented with a page to add, modify or remove existing settings, including changing a setting level (basic/detailed/advanced); the display ordering; settings description, display name, valid values, and data type definition.



Remember that before adding a new key that will be dynamically managed, **you must add the appropriate field to your Settings class, and recompile your host**. The settings you define are mapped to fields in the Settings class; hence these fields must be present first. ConfigWeb allows you, in the **Define/Edit Keys section,** to map a new setting to an existing field name in your Settings class. You can create new settings groups at any time simply by deselecting Use Existing Groups, and entering a new group name. The new group name will be created automatically.



Adding Validation Logic to Custom Configuration Settings

You can add validation logic via the ConfigurationActions class. Simply override the base class validation method as follows:

```
public override bool validateConfigurationKey(object settingsInstance,
ConfigurationKeyValues configurationKey)
```

Always make sure to call the base method first in your override method, as follows:

```
base.validateConfigurationKey(settingsInstance,configurationKey);
```

This method returns true or false, depending on whether the setting passes the base class validation; which validates setting changes to the common host-level settings; and also does type-validation on custom settings you define. The `validateConfigurationKey` method is always called prior to persisting any change into the configuration database.

Storing Custom Classes/Types in the Repository

You can now store large string sequences, including XML/HTML in the repository and associate them with Settings fields. This might include serialized data for constructing custom types/classes from the repository on host startup, and the ability to use ConfigWeb (or programmatic Configuration Service calls at runtime) to update these values and keep running nodes in sync with the updated values. If you will be de-serializing custom types (as defined in static fields within your Settings class); you will need to provide the de-serialization logic. This can be done by overriding the method:

```
public virtual bool updateConfigKeyInHostInstance(ConfigurationKeyValues
updateKey, object settingsInstance)
```

in your ConfigurationActions class. In this method, that will be called for every Settings key defined (including inherited application server global settings every Configuration Service Host has), you must detect the name of the field via the:

```
updateKey.ConfigurationKeyFieldName
```

property. You can use a simple switch statement, and call the base class method for any standard fields not requiring de-serialization:

```
public override bool updateConfigKeyInHostInstance(ConfigurationKeyValues
updateKey, object settingsInstance)

{

      switch(updateKey.ConfigurationKeyFieldName)
      {

            case "myCustomClassField" :

            {

                  Settings.myCustomClassField =
                  deserializeMyCustomClass(updateKey.
                  ConfigurationKeyValue);
                  break;

            }

            default:
            {
```

```
                base.updateConfigKeyInHostInstance(updateKey,settingsInstan
            ce);
            break;
            }

        }

}
```

Also, when setting up Settings fields (configuration keys) that will have large strings as values, you can choose data type "**nodisplay**" in ConfigWeb to prevent display of the large strings except in multi-line edit windows on the Configuration update page.

Executing Logic on Configuration Changes to Custom Settings

The ConfigurationActions class is also used to execute any custom logic you might want to execute when any setting is changed. This logic will be executed on all clustered nodes by default. Some settings might require such logic to be executed when they change, vs. just being updated in-memory via reflection (the default behavior). A good example is in the SimpleTestHarness, TestHost project; changing a background color of the Windows application (one of the custom settings defined for the SimpleTestHost) required a Windows API call to actually make the change live on already-running nodes.

To capture changes as they are made (via ConfigWeb or ConfigurationService programmatic calls), override the checkChangedSetting method as shown below:

```
public override bool checkChangedSetting(object settingsInstance,
ConfigurationKeyValues updatedConfigurationKey, bool block, ServiceUsers
csUser)
```

As with the validateConfigurationKey method, always call the base class implementation first:

```
success = base.checkChangedSetting(settingsInstance, updatedConfigurationKey,
block, csUser);
```

All of the samples, including StockTrader elements, demonstrate using both **validateConfigurationKey** and **checkChangedSetting** method overrides.

Securing Configuration Service Endpoints

Configuration Services can link; providing the ability to configure multiple service hosts making up a composite application from a single ConfigWeb session. For administrative operations, linkages are predicated on an administrative userid/password that is also setup in the repositories of the linked service host(s); and WCF security either at the transport or message level (or both) based on WCF-provided service security, if implemented for Configuration Service endpoints. You should **always secure your Configuration Service endpoints** if you are exposing them externally; or for linked configuration capabilities. Configuration Service endpoints are secured based on WCF bindings that can be easily configured with message + transport or transport-only security (such as https). You will typically want to use **transport security** with your Configuration Service endpoints for this release of the

Configuration Service, given the use of Username client credentials by default, and the fact username/password information is passed over the wire on initial login from ConfigWeb and on subsequent operations between linked services. With transport security, information passed between connected Configuration Services is encrypted. You should refer to Windows Communication Foundation Security on MSDN for a variety of important information and technical guides on implementing secure services.

You can also turn on/off a host-wide setting that will disable any user but the **localadmin** user from viewing or changing configuration data for a host; disabling linked configuration capabilities for that host. This setting is the **ShareServiceConfigurationData** setting. When set to false, configuration data will not be passed between remote hosts, even when authenticated.

Navigating ConfigWeb

When logging into ConfigWeb, the home page will display a root node on the left, and remote connected service domains on the right. If you have administrative rights to remote connected services as well as the ability to connect via WCF message and or transport level security, you can select remote connected services as the root node (much like opening a subfolder in Windows). **You will need to make sure to add the remote service domain's hostname identifier to the Allow Linked Services list setting using ConfigWeb, it is a host-level inherited setting. Until then, the remote service domain will show as non-selectable.** Top-level menu items in ConfigWeb are always operating against the root node you logged into on the login page; however, some items circled below, such as the link buttons, operate on the ***currently selected*** root node (which is always on the left, and can be changed).

Using the Provided CustomUserNameValidator and CustomCertificateValidator Classes

The configuration service provided base classes that enable you to more easily implement custom user name validation (message security with client credentials set to Username); and custom client certificate validation (message security with client credentials set to certificate). These are provided base classes, and you can override the base methods (or modify/extend the base class source code itself) to customize your **Validate** method behavior. The provided **CustomUserNameValidator** works by default with the Configuration Service Users table in your host's repository; but you could (via the method override) easily change this to work with your own user's store. Using the SQL Membership provider is also an option with WCF services, of course, including role-based security; although not illustrated in .NET StockTrader 5.0; a near-term update is planned to illustrate this security option as well in the sample, which the Configuration Service 5.0 does support.

The StockTrader Business Service shows a good example of using message security with the **CustomUserNameValidator** when reconfigured to work with message level security between the StockTrader Web application and the BSL; the *.NET StockTrader Setup and Configuration Guide* discusses how to reconfigure StockTrader to this mode, step-by-step.

The StockTrader Order Processor Service shows a good example of using the **CustomCertificateValidator** class to accomplish customized message level security between the BSL and

the Order Processor Service (OPS); authenticating using client certificates but restricting access to just the BSL's client certificate as a trusted certificate. The *.NET StockTrader Setup and Configuration Guide* discusses how to reconfigure StockTrader to this mode, step-by-step.

Adding Host-Side and Client-Side Binding Configurations

The Configuration Service includes a template set of Configuration Service, Node Service and Primary Service (business service) binding configurations, which are generated into the template configuration file to be embedded in your host process when using the Repository Creation tool provided. While these binding configurations will get developers jump-started quickly, most developers will want to and need to define their own binding configurations with their custom security settings. You can extend the template configuration file within your host with any number of such binding configurations, **but you need to follow a simple naming convention for them to properly appear in the ConfigWeb dropdown selections**. The naming convention separates out bindings meant to be used for hosted services, vs. bindings meant to be used when connecting to remote services as a client.

Host binding configuration names should be prefaced with Host_. For clarity, the name of the binding type should also be contained in the name; for example:

Host_NetTcpBinding_MySpecialService (no security)

If a security mode is applied, this too should be part of the name:

Host_NetTcpBinding_MySpecialService_M_Security (for message security)

Host_NetTcpBinding_MySpecialService_T_Security (for transport security)

Host_NetTcpBinding_MySpecialService_T_Security_M_Credential (for transportwithmessagecredential security)

If you follow these rules, mappings will be made within ConfigWeb, since the Configuration Service reads and understands the bindings present in the host (or remote host's) configuration file. These mappings will automatically pare down selections within dropdown lists when defining service endpoints, making it more straightforward in ConfigWeb to select only matching settings.

For client-utilized binding configurations, the same naming convention applies, except "**Host_"** should be swapped for "**Client_"**. The Configuration Service actually uses client/endpoint configuration sections (that can be easily generated by svcutil.exe) as an indirection to the actual binding configuration used; however, this naming convention for client-utilized binding configurations will make management of binding configurations easier by implying their intended purpose. Hence, you would have:

Client_NetTcpBinding_MySpecialService (no security)

If a security mode is applied, this too should be part of the name:

Client _NetTcpBinding_MySpecialService_M_Security (for message security)

Client _NetTcpBinding_MySpecialService_T_Security (for transport security)

Client _NetTcpBinding_MySpecialService_T_Security_M_Credential (for transport withmessagecredential security)

Always Define Client Configuration Sections and Client-Side Bindings In Your Host's Application's Configuration File for Hosted Service Endpoints

It is very important to understand that your host application's configuration file should have both the host bindings, and the client binding configurations and client configurations (which point to the client binding configuration names) embedded. ***All hosts are also clients to the same services they expose, as they perform online status checks for online monitoring of endpoint status via ConfigWeb.***

You might use svcutil.exe from the Windows SDK to generate your client configuration sections and client-side bindings. This is fine. However, rename the client configuration names and the binding configuration names to follow the "Client_" naming conventions discussed above. This is true for clients connecting to *any* service, whether the remote service implements the Configuration Service or not; as long as your client application is implementing the Configuration Service. You will also want to make sure the contract name referenced in the svcutil.exe-generated endpoint definition is modified to match the actual service contract your client applications uses, including the contract name qualified with the namespace within which it resides.

Modifying Node and Configuration Service Template Bindings

You can also modify or add binding configurations for the Node and Configuration Services, beyond those provided in the template configuration section. These must be prefaced with "**Host_NodeSvc_**" and "**Host_ConfigSvc_"** respectively in the binding configuration name, for them to properly be utilized by the Node and Configuration Services, and appear properly in ConfigWeb selections. As with Primary Service binding configurations, adding the binding type to the name is also typically helpful, although not required. Adding the security mode to the name is required, per Primary Service binding configuration names. This ensures ConfigWeb menu items are trimmed properly to make selections easier to manage later on.

Securing Configuration Service Endpoints

To secure Configuration Service endpoints, you will simply use a custom-defined binding configuration with your desired security mode, and embed this binding configuration in your host application's configuration file (web.config or exe.config). Make sure to follow the naming convention as documented above. You will also need to embed in ConfigWeb's web.config a proper Client Configuration section. This can be generated via svcutil.exe; or simply hand-embedded as desired. See the notes on Adding Client Configuration sections below; as Client Configuration names must also follow a specific naming convention. You will then simply use ConfigWeb to delete non-secure endpoints to the Configuration Service Virtual Host, and add secure endpoints using the newly defined binding configuration. You can also do this for Node Service endpoints, as desired.

Adding Client Configuration Sections

For applications implementing the Configuration Service that will be utilizing remote services, a similar convention for naming is followed. First, you MUST define a client configuration section for each endpoint your application will use. This specifies the client contract and the client binding configuration name the client proxy will use. The endpoint is just a dummy endpoint; as the Configuration Service automatically manages the endpoints via the repository and the dynamic notification capabilities. See any of the provide samples, looking at the configuration file <system.ServiceModel> <clients> section for examples. The client configuration names should be the same as the client binding configurations they refer to for clarity (although they do not have to be):

Client_NetTcpBinding_MySpecialService (no security)

If a security mode is applied, this too should be part of the name:

Client _NetTcpBinding_MySpecialService_M_Security (for message security)

Client _NetTcpBinding_MySpecialService_T_Security (for transport security)

Client _NetTcpBinding_MySpecialService_T_Security_M_Credential (for transportwithmessagecredential security)

These client configuration sections will reference a similarly-named Binding Configuration name to be used by the client.

Generating Client Binding Configurations and Client Configurations with SVCUTIL.EXE

A good way to auto-create the client configuration section and binding configurations clients will need to connect to remote services is with the **Windows SDK svcutil.exe** tool; which will do so for services supporting the Metadata Exchange standard. You will simply need to modify the generated **output.config** so that the generated client configuration name and generated binding configuration name has a proper name (according to the naming conventions discussed above); and the contract name is matched to the actual client contract you are using, including the namespace. Again, see any of the samples for examples. You then simply embed the generated configuration information into the appropriate sections of your client application's configuration file. To emphasize, however:

1. Svcutil.exe-generated output.config binding configurations should be modified according to the naming conventions above.
2. Svcutil.exe-generated output.config client configurations should also be modified according to the naming conventions above.
3. Make sure your <client> definitions map to the renamed binding configuration names.
4. Make sure your <client> definitions map to the correct client contract name (svcutil.exe generates a contract in the generated proxy; but if you are using the Configuration Service-generated client; this contract name may be different. On the client side, the service contract name(s) are the interface names (qualified with namespace) you pass into your client host's startup procedure, per the tutorial.

Adding Service Behavior Configurations

Service behaviors will typically be added as configuration sections within your host's configuration file, as per normal WCF practice. You can optionally supply code-based service behaviors on startup of the MasterServiceSelfHost or MasterServiceWebHost instances. However, it will likely be easier to simply define service behaviors in your configuration file. You will reference these when defining Virtual Hosts within ConfigWeb, as the behavior configuration will be applied to the Virtual Host. The added service behavior configuration sections will appear in dropdown selections in ConfigWeb.

Adding Endpoint Behaviors on the Host-Side

To add an endpoint behavior configuration on the host-side, you will simply add the endpoint behavior configuration to your host's configuration file, and reference it in the appropriate ConfigWeb dropdown when defining a service endpoint in ConfigWeb. You can optionally supply a list of code-created endpoint behaviors as well, on startup of the MasterServiceSelfHost or MasterServiceWebHost instances.

Adding Endpoint Identities on the Host-Side

If you need to specifically assign an endpoint identity to a hosted service endpoint, you will need to use the Configuration Service custom `<endpointIdentities>` section of your configuration file. The handler for this custom configuration is pre-provided with the Configuration Service. An example section (commented out) is provided in every sample in the kit, near the top of the configuration file. The endpoint identities assigned in this section will show up in ConfigWeb in the Hosted Service definition page, which is used to create endpoints. You can use the standard identity parameters (DNS, SPN, UPN, Certificate, etc) within this section.

Adding Endpoint Behaviors on the Client-Side

Endpoint behavior configurations will be generated by svcutil.exe as appropriate for the remote endpoint. Since the Configuration Service creates ChannelFactories that use client configuration sections, these will be respected on Channel creation.

Adding Endpoint Identities on the Client-Side

Endpoint identity information will be generated by svcutil.exe into the client definition section automatically. These will be respected when ChannelFactories/Channels are created.

Naming Your Hosted Service Endpoints

When creating service endpoints in ConfigWeb, one of the most important selections is the friendly name you assign the endpoint. This friendly name will be exchanged via the Configuration Service when clients establish Connected Service definitions to your endpoint(s). These names will be used in code on the client-side when creating instances of the Client class that is the proxy (the Client class definition can be easily generated via the Repository Creation tool). Of course these names will be ignored by service clients that do not implement the Configuration Service, but are utilized by client applications and services that do implement the Configuration Service.

Creating Virtual Hosts and Service Endpoints for IIS-Hosted Services

When you are hosting in IIS, for each primary business service (or node cache service) you create, you will also need a corresponding entry .svc file added to your web application. The .svc file always uses the same one-line syntax, pointing to the same factory class for all services. This factory is always your ConfigurationActions class. The service name in this file maps the .svc file to the virtual host you establish for your service. For example, if your virtual host is "My Service", the .svc file might be named "myservice.svc"; it would look as follows:

```
<% @ServiceHost language = "c#" Debug="False" Service="
TutorialClient.ClientConfigurationImplementation"
Factory="TutorialClient.ClientConfigurationImplementation.ConfigurationActions" %>
```

Service endpoints, when added via ConfigWeb, will always then need to map to the virtual directory of the host application + the .svc filename. For example, if you are hosting in a virtual directory "/myhostapp"; the virtual path for any endpoints to My Service virtual host would be: myhostapp/myservice.svc. The port would typically be one of 80 (http); 443 (https); 808 (for IIS 7 net.tcp endpoints).

You can add relative addresses via ConfigWeb as well. Relative addresses are always appended to the base address for the networl scheme, and there is always a single base address per network scheme (http, https, net.tcp, etc.). So, if you have a base http endpoint address for My Service; it would be http://myhostapp:80/myservice.svc. If you add a second http-based address with a virtual path of "address2", it would be http://myhostapp:80/myservice.svc/address2

Establishing Connected Service Definitions to Remote Services

For client applications and/or services that implement the Configuration Service, you will use ConfigWeb to establish Connected Service definitions on a per endpoint basis. These 'subscriptions' then become dynamically managed by the Configuration Service. There are two types of remote services you might connect to: those that are hosted by a host process that also implements the Configuration Service, and those that are hosted by a process that does not implement the Configuration Service (Java and .NET inclusive). These are referred to **as Primary Services** vs. **Generic Services** in ConfigWeb and the documentation.

Before establishing a Connected Service definition, your client's host application solution (.exe, Web application or NT Service) must have a valid client contract (perhaps generated by svcutil.exe); data contract (also perhaps generated by svcutil.exe); and the appropriate client configuration section with it's appropriate referenced binding configuration (also perhaps generated by svcutil.exe but renamed according to naming conventions discussed above) already present in the client application's configuration file. The client contract(s) for connected services must also be passed in to the MasterServiceSelfHost or MasterServiceWebHost classes as part of the constructor. You can reference the provided samples or go through the tutorial to see how this works.

Connecting to Primary Remote Services via ConfigWeb

Using the **RemoteSvcs** link button in ConfigWeb, you will simply specify an endpoint to an online instance of the remote host's Configuration Service, and click **Get Services**. Again, do not do this until you have satisfied the requirements:

a) Client contract(s) already part of your client application's solution as compiled/running.
b) Client contract(s) properly passed into initialization of your MasterServiceSelfHost or MasterServiceWebHost class instance.
c) Client configuration section generated, properly re-named according to the naming conventions, and embedded in your client application's configuration file in the <system.ServiceModel><clients> section.

Assuming these steps have been completed, you will just point at the remote host's Configuration Service in the Connected Service page in ConfigWeb, and this will retrieve the metadata necessary to establish a subscription to one or more of the service endpoints hosted by the remote host. You will want to make sure the client contract and client configuration name is properly matched in the dropdowns, however.

Connecting to Generic Remote Services via ConfigWeb

A Generic Service is a service (.NET or other platform) hosted within a host process that does not implement the Configuration Service. The process is similar, however a bit more information needs to be supplied. You will need to enter a Host Name Identifier, and assign a service-friendly name explicitly. You can assign multiple service subscriptions (endpoints) to the same Host Name Identifier—this would be done if multiple service endpoints are hosted by the same host application server, for example, so they are grouped together in various ConfigWeb pages. The connected service username/password you enter will simply be created and used on the client side; and again, the same user name should be assigned to service endpoints hosted by the same assigned Host Name Identifier.

The assigned service friendly name, as with Primary Services, will be used when instantiating your service client. The actual WCF service client code (a base class) is no different for a Generic Service vs. a Primary Service.

Changing Hosted Service Definitions Once Clients are Already Subscribed to an Endpoint

Changing certain fields on the host side for endpoint definitions (via the Hosted Service page) in many cases will automatically end client-side subscriptions and remove established Connection Points. This is done through notifications that flow between hosts and clients that implement the Configuration Service, by design. Such changes would break clients, and hence the subscriptions are ended automatically (as is possible) such than an administrator/developer must explicitly re-establish them to ensure client applications continue to work. So be careful when changing information for established services; even if clients cannot be notified to end their subscriptions and remove existing Connection Points, many changes can break existing clients (such as changes to service contracts; data contracts; address paths; ports; etc). It may be better to version these endpoints by creating new Virtual Hosts

and new service endpoints while still running the old endpoints, depending on the nature of your connecting clients and whether these are all deployed within a controlled service network.

In the future, more sophisticated logic in the version notifications might be incorporated into the Configuration Service, rather than just ending client subscriptions. However, explicitly ending these Connected Service subscriptions via notifications was deemed better than just leaving broken subscriptions on the client. If connected clients cannot be notified (all nodes are offline when a change is made); then of course the administrator of the client application can explicitly remove/re-establish the Connected Service definitions via ConfigWeb. In general, whenever a notification to a remote connected client or remote connected host is performed, there is also a failsafe manual mechanism to perform the operation via ConfigWeb if it cannot be performed automatically.

Dynamic Clustering for Load Balancing and Failover

There are some important considerations when utilizing the load balancing built into the Configuration Service. First, understand that the load balancing and failover is based on the use of an intelligent WCF client that can be generated via the Repository Creation tool given an existing service contract (interface) + service data contract assembly. These contracts might be under your control (e.g. provided directly in the solution as per the service host); or generated via svcutil.exe for services you connect to that are outside of your development domain[1].

The first is that not all operation errors can be avoided when a network or remote host crashes. Online checks are performed before the customer client code is handed a communication Channel; however, for any service operation in-process during a crash (request made but response not yet generated/received); proper exception handling is still crucial, as the client code will see these exceptions, even as future requests are automatically redirected to other online nodes. This is simply the nature of remote service calls: only the developer can fully understand what exception handling to employ when a request is made but a valid response is not received. Should the operation be re-tried? Should the information be stored for an audit log? This is up to the developer of the client application. So, you will see exceptions for clients that are actively in the process of waiting for a response from a remote host when it crashes or the network goes down. These should be few, even under heavy load, as the number of Channels actively being used at one time is generally fairly limited, and also constrained by the number of worker threads in the client application's thread pool. Future requests to

[1] Please remember that for a SOA deployment, it is perfectly valid to share contract (schema) between clients and hosts; but never implementation logic which is black-boxed. So, if you control the development-side of both the service host and the service client, there is no need to run svcutil.exe to generate the service contract and data contract for a client! You can simply use the projects or compiled assemblies you are already using on the host-side for these contracts within your client Visual Studio Solution, since this is *only* schema information, not implementation logic. This can be preferable, at times, and even should be encouraged for cases where you are developing both the client and the service (or both are developed within your organization), as certain generic types can be treated in .NET-capable ways within your client (svcutil.exe tends to convert generic Lists to arrays, for examples, to adhere to industry standards for non-.NET clients). Note that this practice does not in any way prevent your client from connecting to non-.NET services that implement the same service contract/WSDL-based data contract, hosted on Java application servers, for example. The StockTrader application is a great example of this.

the downed node will automatically be redirected to other online nodes so they will not receive exceptions on remote calls, assuming other nodes are still online.

Removing Downed Nodes Manually

As the administrator of a service host domain, crashed nodes will be immediately noted in ConfigWeb UI (SOA Map and View Nodes pages). In general, crashed nodes are never explicitly removed from the repository or intra-node communication channels, by design, because they might come back online at any moment, assuming just a temporary network disruption. Crashed nodes can be restarted, and the Node Service will continue working. However, if you have a node that crashed and cannot be restarted, or you do not want to restart, you can manually remove it from the intra-node communication service and the repository via the **View Nodes** page in ConfigWeb. A link to remove a node that is not responding will appear, if the node still has an entry in the repository (true only if the node truly crashed, vs. the process being ended gracefully via a close (.exe) or worker process IIS stop (web hosted services).

Removing Clients Manually

Client connections are maintained by the Configuration Service dynamically, even as new nodes on connected clients are started. You can monitor the client applications (that do implement the Configuration Service) that are active via ConfigWeb Connections page. Clients are typically never explicitly auto-deleted from the host's repository; by design (unless an administrator action for the client service domain makes such a request). However, as an administrator of the host service domain, you can use ConfigWeb to explicitly remove service client connections as desired, via the Connections page. You might do so, for example, if a client is noted as down for a period of several days; or you also control the client service domain and know a client node will not ever be reactivated. Remember, if the client node is reactivated, it will still automatically rejoin as a connected client, as before.

Load-Balancing Against Java Application Server Services or other Non-.NET/Non-Windows based Hosts

Load-balancing built into the Configuration Service requires that the client application implements the Configuration Service. It can load balance/failover against both .NET and non-Windows based service endpoints, however (and .NET endpoints hosted within hosts that do not implement the Configuration Service). If you are building a service host that has clients that potentially do not implement the Configuration Service (for example, cloud-based http protocol services with an unknown/uncontrolled number of clients), you can integrate any endpoint with either Windows Network Load Balancing, or hardware load balancers such as Cisco CSS and F5 BIG-IP. You do so when establishing your service endpoints via ConfigWeb.

For .NET-based clients implementing the Configuration Service that connect to services that also implement the Configuration Service, endpoint management is completely dynamic based on notifications. For Java-based hosts (IBM WebSphere, BEA Web Logic, Oracle Application Server, etc.); or for .NET/ASMX services that do not implement the Configuration Service on the host-side, you simply can define as many Connection Points to a service, on as many nodes the service is running on. All the

same status checks/failover is performed if multiple Connection Points are defined to the same service, but for different physical servers (nodes).

Polling Logic

If a client attempts to establish a WCF Channel to a node that is offline, and fails, that node will be added to a polling list. Such downed nodes are polled every 60 seconds on background threads; such that they will be added back into the cluster definition for load balancing from connected client nodes within 60 seconds should they come back online. This is true of .NET nodes implementing the Configuration Service, and .NET or Java-based nodes that do not implement the Configuration Service. The polling logic is automatic, such that temporary network disruptions do not permanently end requests to a network-disrupted node for more than 60 seconds should the network connectivity become active again; or the service host be started up again by the administrator of the service host domain (or host rebooted, etc). As an example, you could start/stop the IBM WebSphere 6.1 application server on a node in a WebSphere cluster, and the .NET StockTrader Web application will again start utilizing that node once the WebSphere application server has been restarted. The key is that you should use ConfigWeb to establish multiple Connection Points to such Generic Services (those that do not implement the Configuration Service) for each node you want to load balance/failover against. This is not necessary when connecting to service host's that do implement the Configuration Service, since notification logic automatically flows between hosts and clients such that all running nodes are automatically added to a cluster definition, and only a single Connection Point need be established to set up this dynamic relationship between clients and hosts.

Remember that persisted connection points within a dynamic relationship are, by design, not removed from either a host or a client's repository should an application node simply be stopped. Rather, the connection point is simply deactivated, until that node starts up again. New client nodes always check every persisted connection point for validity on startup; polling will occur against service nodes expected to be online, but found not to be online. Hosts never poll clients; this is not necessary since communications to clients via the connected Configuration Services occurs only during certain constrained Configuration Service operations; and no in-memory dynamic list of connection points needs to be maintained—they are simply read from the repository when such a notification needs to take place; with full handling of communication exceptions should any client nodes be offline. Should you want to permanently remove a connection point (either client-side or host-side); you can do so at anytime using ConfigWeb, as covered earlier in this document, via the Connections menu item.

Creating Service Endpoints that Listen on an Explicit IP Address or DNS Name

You can create service endpoints that are constrained to listening on just a pre-provided address vs. listening on all IP addresses configured on the node. To do so, you will use the configuration file to provide an <appSettings> key that consists of the nodes DNS long name + "==" + the hosted service name to be constrained, with a value equating to the specific IP address or alternate DNS name to listen on. Note, the binding configuration used in such a setup should have the **HostNameComparison** setting set to **Exact** vs. **StrongWildCard** for this to work with WCF. The name you listen on might simply be the IIS Host Header, for example; or an externally exposed address. You can add into configuration a list of nodes in this fashion (via their DNS long name); such that configuration files remain the same on all

nodes; but nodes find their own node-specific IP addresses for the specified services. This should only be done with **base addresses**! See the SimpleServiceHost for an example with a commented section in its app.config configuration file. If you constrain an address in this fashion; your specified address(es) will be used in all cross-domain notifications; vs. the default behavior of using the long host name of the server.

Creating Multiple Virtual Hosts within One Master Process

You can create as many Virtual Hosts (ServiceHosts or WorkflowServiceHosts) as you want within a master host process. You are not constrained to creating just one; however consideration should be given to partitioning services in an intelligent way to best support data center operations and ongoing maintenance. It may be preferable, for example, to create two distinct service domains each with their own service repository, vs. creating multiple virtual hosts that live within the same host process and share the same repository. This design decision will be scenario specific. Just be aware that creating multiple virtual service hosts within the same master host process is permitted, as the *MoreInterestingTestHarness* sample in the download demonstrates (see **ServiceAHost**).

Creating Services that Implement Multiple Contracts

It is also perfectly permissible to create a single service implementation class that implements multiple service contracts, which is also demonstrated in the *MoreInterestingTestHarness* sample. Clients will see these as two logically distinct services with their own endpoints; and ConfigWeb will allow clients to subscribe to any of the implemented contracts and their endpoints, as separate services.

Using the Repository Creation Tool

Please note the Visual Studio 2010 template replaces the standalone Repository Creation tool for creating a base implementation. However, the Repository Create tool still is used to generate service clients that inherit from the base load balancing with failover client class. With the Configuration Service 5.0 Template, the manual steps for creating a base implementation for all host types have been completely eliminated.

Generating and Using the Client Class

In the Configuration Service 5.0, new functionality has been added to the Client generated by the Repository Creation tool. In version 2.x, Channels (and ChannelFactories) were created automatically. These Channels were always shared across all operating threads, for performance reasons. This is still the default behavior using the Client class constructor:

- `TheClientClass client = new TheClientClass(string serviceFriendlyName);`

And this constructor should be used for normal multi-threaded operations to connected services as long as all client instances will have the same ChannelFactory and Channel characteristics. However, there are cases (such as when using DuplexChannels with InstanceContext; or setting username credentials on a per request basis) when client code needs to ensure the Channel used is unique to just that request (or series of requests). Hence, you can now use the following new constructors:

- `TheClientClass client = new TheClientClass(string serviceFriendlyName, bool createNewChannelInstance)`
- `TheClientClass client = new TheClientClass(string serviceFriendlyName, bool createNewChannelInstance, InstanceContext instanceContext)`
- `TheClientClass client = new TheClientClass(string serviceFriendlyName, bool createNewChannelInstance, string userNameCredential, string passwordCredential)`
- `TheClientClass client = new TheClientClass(string serviceFriendlyName, bool createNewChannelInstance, InstanceContext instanceContext, string userNameCredential, string passwordCredential)`

If the createNewChannelInstance parameter is true, then a **unique** new Channel instance will be created, and all operations against the TheClientClass.Channel property will be through this unique instance, and not through the separately maintained shared Channels (which still exist). In such cases, the created Channel will be based on a round-robin list of service host nodes known to be online at the time the constructor is called; and subsequent requests through the client instance will be directed at the same selected host node (affinity).

When creating new Channel instances via the createNewChannelInstance parameter on the client class constructor, it is very important that after using the client instance, you call client.Close() as this will close out the Channel and ChannelFactory (or abort it if faulted); cleaning up resources and freeing connections from the pooled connections that exist within the .NET network stack.

Failure to call Close() will result in rapid depletion of available network connections when running under load, and poor/degrading performance. You can also call client.Close() even when not creating new Channel instances, as the Close logic is smart enough to just ignore such calls. So: it is a good idea to get into the habit of calling client.Close() after using a client instance, whether the client is using unique Channel instances or the default shared channels.

When using Duplex contracts, you must always create a unique channel instance and pass in InstanceContext. When passing in InstanceContext, DuplexChannelFactories are automatically created instead of the standard ChannelFactories.

Important Callbacks on ChannelFactory Creation

The Configuration Service creates Windows Communication Foundation (WCF) channels to remote services (whether based on .NET, Java or other non-Microsoft platform) and manages these Channels within the infrastructure. The base Client class provided is specifically designed for client operation endpoint management, load balancing and failover across the major WCF bindings. The default behavior is to use a set of cached channels, such that the expense of creating new proxies is reduced. Channels are created by automatically-instantiated ChannelFactories, such that developers do not need to deal with these lower-level details of WCF client development, and channel lifecycle management is handled by the infrastructure. However, there may be cases where developers want to customize ChannelFactories for all or specific service endpoints, prior to any channels being created from the ChannelFactories. This is enabled through the base class methods:

- `afterCreateChannelFactory`
- `afterCreateChannelWithUserNameCredentials (used when the Connected Service Definition has default Username credentials specified)`
- `afterCreateDuplexChannelFactory`
- `afterCreateDuplexChannelWithUserNameCredentials (used when the Connected Service Definition has default Username credentials specified)`

These methods, contained within the ConfigurationActionsBase class, can easily be overridden, such that customization of any ChannelFactory can be accomplished, even for the infrastructure-handled cached ChannelFactories/channels they create. Simply override these methods in the host's ConfigurationActions class implementation; modify the ChannelFactory, and return it to the calling method. You can detect what endpoint is being created based on either the `clientConfiguration` or `serviceFriendlyName` that are parameters to this call; and customize the ChannelFactory for specific service endpoints (such as setting client credentials; removing default context, etc.).

These methods are called not only for Primary Business Services, but also for the Configuration and Node Services; hence can be used to modify the Channels created for Configuration, Node and Node Distributed Cache (DC) operations.

Important Calls to Create and Manage Your Own ChannelFactories and Channels

The methods presented above operate against ChannelFactories that are by default cached and used against load-balanced service endpoints, for horizontal scalability and per-operation level failover. However, developers are not restricted from using these pre-created, cached Channels. There may be cases where developers need to manage their own ChannelFactories and Channels, but want to use the dynamically managed node endpoint information for load balancing and failover, just like the cached channel sets provide. This is possible as well, with a set of APIs the base Client class supports. The *MoreInterestingTestHarness* sample demonstrates this, as there are programmatic calls you can make to retrieve endpoint information for these purposes; and yet still be returned only online endpoints, and endpoints that get load balanced between such API requests (in other words, round robin is used to lookup and return the endpoint information on each separate request for load balancing; and the endpoint is checked for validity before being returned). These calls include:

- `getANodeAddress`
- `getANodeEndpoint`
- `getAllOnlineAddresses`
- `getExistingChannelByAddress`
- `createANewChannelFactoryByAddress`
- `createANewChannelByFactory`

All of these calls will only return valid, online endpoint instances. You can use such calls, for example, to create/manage your own ChannelFactories while still taking advantage of load-balancing/failover. The *MoreInterestingTestHarnness* demonstrates this with a WorkflowServiceHost.

Important Callbacks after ServiceHost and WorkflowServiceHost Creation

If you need to modify the Description of a ServiceHost or WorkflowServiceHost (defined as a Virtual Host across running nodes) prior to it being opened, there are two important callbacks that can be overridden to perform customization of the Description, or perform any other application initialization logic. Both are contained in the ConfigurationActions class, which will be part of your HostConfigurationImplementation project (see tutorial). These methods are:

- `afterCreateServiceHostFromRepositoryBeforeOpen`
- `afterCreateWorkFlowServiceHostFromRepositoryBeforeOpen`

You will simply need to override these virtual methods of the abstract base ConfigurationActionsBase class, and can then use the provided ServiceHost or WorkflowServiceHost instance (that come in as a parameter and are expected to be returned after modification) to make modifications to the Host Description. For example, you might conditionally remove existing repository-defined endpoints, add new service endpoints on a conditional basis, add/remove service or endpoint behaviors, etc. You can also use these methods to perform initialization logic not related to the ServiceHost/WorkflowServiceHost itself, but logic that needs to be executed before the node opens the ServiceHost/WorkflowServiceHost to start receiving service operation requests. You can look at the StockTrader samples or other samples provided for examples of overriding these methods with your own service hosts. These callbacks are functional for both IIS-hosted services and self-hosted services.

You could use the code below, for example, to get the WorkflowRuntime such as:

```
WorkflowRuntimeBehavior workflowRuntimeBehaviour =
host.Description.Behaviors.Find<WorkflowRuntimeBehavior>();
WorkflowRuntime rt = workflowRuntimeBehaviour.WorkflowRuntime;
```

For IIS-hosted services, including both ServiceHosts and WorkflowServiceHosts, the following callbacks are also functional:

- `applyConfigurationServiceHost`
- `applyConfigurationWorkflowServiceHost`

These methods can be used to intercept WCF-infrastructure logic that applies host configuration information from the web.config file. These simply are called at the same point in the lifecycle that the ServiceHostFactory `applyConfiguration` methods are called.

In reality, the MasterServiceWebHost is underneath a ServiceHostFactory that is fully tied into the Configuration Service repository and runtime management logic.

Working with Duplex Contracts

New with the 5.0 release is support for Duplex Contracts and the wsDualHttpBinding. There are some special considerations when adding Duplex services as hosted services; as well as for client applications implementing the Configuration Service that will be clients to Duplex services. These are minor, and examples are highlighted below for both the host side and client side applications. Namely, Duplex

contracts require that InstanceContext be provided for all client calls. You will still implement an isOnline method in your Duplex contract for online status checks:

```
[OperationContract(IsOneWay = true)]
void isOnline();

public void isOnline()
{
    return;
}
```

1. On the **host side**, you must use a new initialization procedure when creating your startupList that contains the virtual host information for all virtual hosts you will be hosting. An example is provided below:

```
List<ServiceHostInfo> startupList = new List<ServiceHostInfo>();
List<ServiceClientInfo> clientInfo = new List<ServiceClientInfo>();
ServiceClientInfo info1 = new ServiceClientInfo(new InstanceContext(new
      CallbackHandler()),typeof(ICalculatorDuplex));
clientInfo.Add(info1);
ServiceHostInfo info = new ServiceHostInfo("Calculator Host", false, null,
new object[] { }, new CalculatorService(),clientInfo);
startupList.Add(info);
```

The highlighted items above are the new elements. A new class, ServiceClientInfo, will be created for each contract you are hosting that is a Duplex contract. You do not need to create a ServiceClientInfo class for non-Duplex contracts. When creating this type, you will need to supply a default InstanceContext of the correct type for the contract, depending on the callback handler contract implemented. Next, you will create a List<ServiceClientInfo> and add each ServiceClientInfo entry into this list as shown above. Finally, when creating a ServiceHostInfo class for a virtual host, simple pass in this list as shown above. You will then pass in your startupList to the initialization logic as usual (not shown) as documented in the tutorial.

2. On the **client side**, a similar procedure is necessary when initializing your host, as shown below:

```
List<ServiceHostInfo> startupList = new List<ServiceHostInfo>();
ServiceClientInfo info1 = new ServiceClientInfo(new InstanceContext(new
     CallbackHandler()),typeof(ICalculatorDuplex));
//This starts the Windows program main form, which is a class that inherits from a
base class with all the Windows logic/display layout pre-provided.
Application.Run(new ServiceHostConsole(new NTService2.TheSettings.Settings(),
     new ConfigurationService(), new
     ConfigService.ServiceNodeCommunicationImplementation.NodeCommunication(),
     null, new ConfigurationActions(), startupList, null, new object[] {
     info1}, null));
```

The only difference is that instead of simply passing in an object array of client service contract types; you will instead pass in an instance of type ServiceClientInfo. The object[] can contain a mixture of ServiceClientInfo types and service contract types (in form of typeof(IServiceContract).

3. Also on the **client-side**, when using the generated client class (generated via the Repository Creation tool); there will be new constructors that allow you to specify unique Channel Instances be created, and the ability to provide a unique InstanceContext instance for each client created. Please read the previous section on **Generating and Using the Client Class**. If your client is based on a Duplex Contract, you will always indicate to create unique Channel instances when creating Client instances; and pass in an InstanceConext that will be unique for that client instance.

Management of Lifecycle for IIS 7.0 WAS-Hosted Services

In IIS 6.0 it is only possible to host http-based endpoints for a WCF service. With IIS 7.0 (Windows Vista and Windows Server 2008), you can host non-http endpoints including tcp (with binary encoding); Msmq; and named pipe endpoints as well. It should be noted that lifecycle management of such endpoints within an ASP.NET worker process is slightly different. Namely, the global.asax `Application_Start and Application_End` events are not fired if the first call into the ASP.NET application hosting the service is to a non-http endpoint. The Configuration Service fully takes this into account, and manages the lifecycle appropriately. Thus, you can use

- `afterCreateServiceHostFromRepositoryBeforeOpen`
- `afterCreateWorkFlowServiceHostFromRepositoryBeforeOpen`

to perform any additional host initialization logic in place of global.asax if you need to. However, realize that these methods will be called for each Virtual Host defined in your host process; hence you can use the `VHostName` parameter to capture a specific ServiceHost (or WorkflowServiceHost) that is being initialized, and only execute this logic once based on that name.

In all cases, please remember that you cannot use global.asax `Application_Start` and `Application_End` events reliably for initialization/shutdown logic if you are WAS-hosting non-http endpoints in IIS 7.0. This is per WCF design, and not related to the Configuration Service. The Configuration Service provides a mechanism to compensate. Interestingly, if the first endpoint activated is http-based, the global.asax events fire as expected.

CustomActions

CustomActions are a way to perform logic across clustered nodes, automatically. The method signature and implementation logic for the CustomAction will be defined in your host's ConfigurationActions class. A CustomAction class will be instantiated that will contain the method name (as you implemented in your ConfigurationActions class), and a pre-set `List<string>` parameter, that can be used to encapsulate any parameter that can be represented as a string, or serialized thereof if you want to construct custom types based on this generic list. The CustomAction is a convenient way to ensure application logic is executed locally on every running node during your application/service lifecycle. You can define as many CustomActions as you like within your ConfigurationActions class, whether they are executed based on Configuration changes or based on other application conditions not necessarily triggered by configuration changes.

Can Nodes be Geographically Dispersed?

Yes. As long as they can connect to the same service repository. You could, for example, host four clustered nodes in one data center, and four in another. This is not true for Azure-hosted services, however, as each will be using a different load-balanced base DNS address.

Is the Configuration Repository a Single Point of Failure?

Yes and no. If your configuration service databases are hosted on SQL Azure, there is automatic high availability with one primary and two replicated SQL Azure databases—this is just a base feature of SQL Azure.

If hosting on-premise with SQL Server, should the service configuration database go down, no Configuration Service operations will be able to be performed, and new nodes will not be able to start. However, Primary Service operations and node notifications (including distributed cache service operations) will still be fully functional, so connected clients will still operate while the repository is down. In an environment built for ultimate reliability, however, you will want to cluster your on-premise configuration database using Windows Clustering Services, which allows for multiple active/passive SQL Server nodes to be defined such that if one node fails, a new SQL Server 2008 instance will be spun up and take over automatically. See documentation on Windows Clustering Services for details. The same technology can be utilized for MSMQ message queues and the Microsoft Message Queue Service.

Node Notification Failures

ConfigWeb will note if configuration updates are not successfully communicated to peer nodes. It will also note how many peer nodes were not successfully notified, potentially because of crashed nodes, or network connectivity failures (bad NICs, network cables, switches etc.). You should investigate such failures immediately to identify and correct the situation (also, see the section on removing permanently downed nodes that have not gracefully closed). You can configure multiple Node Service endpoints and have these endpoints listen on different (explicit) NICs/IP addresses. You can mark a passive Node Service endpoint as a failover endpoint, such that if a switch, network cable, etc. fails on a node, other nodes automatically use the failover endpoint for Node notifications. Node notifications are automatically retried even without such a failover endpoint.

For Configuration Service 5.0, node notifications are not rolled-back should a single node be unresponsive/fail the operation. Ultimately, if all Configuration Service operations behaved in such a fashion, a single downed node would result in all Configuration Service operations from being able to complete until the downed node is either removed from the cluster (via ConfigWeb), or successfully restarted. Future updates may include the ability to designate certain operations as transacted, and just for these operations, distributed transaction processing performed such that all expected online nodes complete the operation, or none do.

Using the Configuration Service Programmatically For Automated Actions

The ConfigWeb UI interacts with the Configuration Service on a user-driven basis, from a Web-based user interface. However, all operations can be driven programmatically as well, as demonstrated in

various samples in the download. Hence, you could use specific Configuration Service operations within an automated process to change configurations, or perform automated notifications to administrators. For example, the getNodeMap service operation could be driven by a background thread, such that if any node was determined to be offline, a text message or email could be generated on the server side to notify an administrator, or even trigger an SMS/MOM-monitored event. In addition, the ConfigWeb user interface is just one possible user interface that could be layered on top of the Configuration Service. Smart client front ends (Silver Light, WPF, etc) could be built as well, although not provided in the current implementation.

ConfigWeb and Service Domain Location Independence

Because ConfigWeb just uses a service-oriented paradigm to connect to hosts (actually **virtual hosts**) that implement the service, it can be staged anywhere that has network connectivity to a root service/application node. It is a completely autonomous application, and can be staged on a laptop and taken mobile with a user, who can then use it to attach to a service remotely from behind a firewall within a remote hotel room, for example, given proper Internet connectivity and authentication to access the remote service host. In this manner, it is also possible to setup ConfigWeb on the Internet, intranet or extranet, and use it to monitor/administer service domains in geographically dispersed regions, assuming Configuration Service endpoints are properly secured using WCF-based transport and/or message-level security (or both!).

The Best Physical Network Setup for Deployed Scenarios

While comprehensive advice for configuring the ideal network topology for a service-oriented deployment is beyond the scope of this document, in general it will be ideal to separate client traffic onto dedicate networks (NICs) that are designated as inbound for connecting clients; vs. operations performed with the Node Service, or operations invoking other remote services; or database connectivity to a primary business service database. Using multiple NICs in a server is encouraged, with DNS (or IP addresses) controlled in such a fashion to utilize the different NICs across different subnets, both for security and performance reasons. For example, you may have your Primary Business Service endpoints and Configuration Service endpoints listening on a specific network (inbound with connectivity to the clients, potentially hosted in the cloud); while your database connections (including for production business-related databases and the configuration database) utilizes a different subnet to connect to the database(s), potentially located on the other side of an additional firewall.

A good example is with StockTrader. Configuring the IIS servers running the Web application (likely running behind a front-end firewall within a DMZ) inbound listen IPs to be on a dedicated client inbound network vs. the remote calls made to Business Services (which would likely be deployed behind a second firewall) configured on a completely different NIC/subnet is ideal. In this way, network traffic can be more balanced across the subnets, and additional security achieved since remote invocations of Business Service operations happen on a separate subnet the clients have no direct access to. It should also be noted that in such a remote setup, the production database is protected outside of the DMZ, operating on an internal subnet to which the business service layer, but not the ASP.NET StockTrader Web application (exposed to the cloud), has private access.