Cloud Service Fundamentals in Windows Azure

This document provides an overview and walkthrough of the Cloud Service Fundamentals (CSFundamentals) application that demonstrates how to build database-backed Azure services. It includes a description of the scenario, implementation architecture and reusable components for logging, configuration and data access. The code base is intended to be a tangible exploration of best practices for delivering scalable, available services on Azure based on production deployments by the Windows Azure Customer Advisory Team.

**Creators:** Michael Thomassy, Rafael Fernandez, Mark Simms, Christian Martinez, Ewan Fairweather, James Podgorski, Paolo Salvatori, Rama Ramani, Shaun Tinline-Jones, Silvano Coriani, Valery Mizonov

# Resources

For additional reading and background, please refer to the following resources:

* [Building Big: Lessons learned from Windows Azure customers - Part I (Scale)](http://channel9.msdn.com/Events/Build/2012/3-029)
* [Building Big: Lessons learned from Windows Azure customers - Part II (Availability)](http://channel9.msdn.com/Events/Build/2012/3-030)
* [Best Practices for the Design of Large-Scale Services on Windows Azure Cloud Services](http://msdn.microsoft.com/en-us/library/windowsazure/jj717232.aspx)
* [Failsafe: Guidance for Resilient Cloud Architectures](http://msdn.microsoft.com/en-us/library/windowsazure/jj853352.aspx)

# Scenario Description

Many modern cloud services integrate social experiences, based on a foundation of registering and managing per-user information. The intent of the CSFundamentals code reference is to demonstrate several best practices and experiences for implementing modern data-driven applications on Azure in context. The code base implements the following functionality:

* [MVC4](http://www.asp.net/mvc/mvc4) web application providing scalable user registration and login against a sharded database with distributed cache integration
* Multi-cloud service deployment, using Application Request Routing (ARR) to transparently leverage multiple cloud services for additional scale and reliability
* Queryable operational data store, with scheduled tasks for collecting and integrating application and server performance/health metrics.

# Visual Studio Project Layout

The CSFundamentals applications incorporates several projects in one solution and three cloud service definitions for use by developers. This section provides an overview of these artifacts, the directory structure and third party libraries.

## Required Visual Studio Environment

The reference implementation was developed using Visual Studio 2012 Ultimate; however, there shouldn't be issues with other editions. The following items need to be installed:

* Windows Azure SDK v2.0. Recommended to install through the [Web Platform Installer](http://www.microsoft.com/web/downloads/platform.aspx).
* [Microsoft® SQL Server® 2012 Express](http://www.microsoft.com/en-us/download/details.aspx?id=29062) (for the sqlcmd.exe utility).
* For SQL Reporting, install [Microsoft SQL Server Data Tools - Business Intelligence for Visual Studio 2012](http://www.microsoft.com/en-us/download/details.aspx?id=36843).

## Visual Studio Solution

The code base is contained in one Visual Studio solution, **CSFundamentals:**

* CloudServiceFundamentals.sln. Open this Visual Studio solution to work with the application code, reusable libraries, and publish the application services to Azure.

## Visual Studio Projects

The overall solution consists of several projects; with reusable and scenario specific code separated.

* Common: Reusable components.
  + Microsoft.AzureCat.Patterns.Common. Reusable baseline aspects of the codebase, including configuration, logging, serialization and helper base classes.
  + Microsoft.AzureCat.Patterns.Web. An implementation of the MVC4 providers and configuration utilities designed to work against a scale-out relational data service.
* Data: Libraries for working with data stores.
  + Microsoft.AzureCat.Patterns.CacheClient.AzureCaching. Provides a wrapper around the Azure Caching client SDK, including the use of a pluggable binary serializer (in this case [protobuf-net](http://code.google.com/p/protobuf-net/)).
  + Microsoft.AzureCat.Patterns.Data.Common. Helper classes for working with data storage, relational and non-relational, including retry and telemetry logic for working with Windows Azure SQL Database.
  + Microsoft.AzureCat.Patterns.Data.SqlAzureDalSharded. Helper classes for implementing a scale-out relational database solution, including partitioning, connection affinity and metadata management.
* Database: Database schema definitions.
  + Microsoft.AzureCat.Patterns.Data.SqlAzure.OpsStatsDB. The schema for the operations database, used to centralize telemetry information from compute, storage and SQL.
  + Microsoft.AzureCat.Patterns.Data.SqlAzure.RootDB. The database project for the user metadata and shard management databases.
  + Microsoft.AzureCat.Patterns.Data.SqlAzure.UserProfileDB. The database project for the user profile and information databases.
* Logic:
  + CSFundamentals.Logic. Application specific logic and data schema.
* Reporting*:*
  + OpsStatsDBReporting
  + Parameterized reports (.rdl) using SQL Reporting from the OpsStatsDB database. The reports show query stats, database size & growth, wait times and errors across all sharded databases.
* Roles*:* Cloud service web and worker roles.
  + CSFundamentalsCache.WorkerRole. Hosting the Azure Caching web roles providing caching for the ConstosoService.WebRole.
  + CSFundamentalsService.WebRole. The public web site exposed by the MVC front-end web application as an Azure Web Role.
  + RouterService.WebRole. The worker role hosting the cookie based application request routing (ARR).
  + SchedulerService.WorkerRole. The worker role hosting the [Quartz.NET](http://quartznet.sourceforge.net/) scheduler.
* Router:
  + Microsoft.AzureCat.Router.LoadBalanceThenCustomCookieProvider. Load Balance requests based on the cookie contents routing traffic to multiple cloud services (multiple CSFundamentalsServices).
* *Tasks:*
  + Microsoft.AzureCat.Patterns.Tasks.SqlAzureMgmt. Collection of scheduled tasks using the Quartz scheduler to collect and load telemetry data from different sources into a Windows Azure SQL Database for exploration and charting.
* *CSFundamentalsService:* Azure web service hosting the front-end CSFundamentalsService.WebRole and CSFundamentalsCache.WorkerRole.
* *RouterService:* Azure web service hosting the RouterService.WebRole.
* *SchedulerService:* Azure web service hosting the SchedulerService.WorkerRole.

## Shared Libraries

The reference implementation makes use of multiple shared libraries, managed through NuGet packages. The Microsoft libraries distributed through NuGet used by this solution include:

* Reactive Extensions (library for supporting asynchronous and push-based operations)
* Enterprise Library’s Transient Fault Handling Application Block

The non-Microsoft libraries used by this solution are:

* Dapper (micro-ORM, used in the database access code)
* Quartz.NET (scheduler library, used for managing recurring tasks)
* Common.Logging (used by Quartz.NET)
* NLog (application logging library)
* Protobuf-net (binary serializer)

# Reusable Components

There are several reusable aspects bundled with the CSFundamentals application. This section will describe each of them, explain their design, configuration and potential applicability to your own projects.

## Configuration

Traditional .NET application configuration depends on the use of System.Configuration pointed to app.config or web.config files. In an Azure service these files are part of the deployed package (and hence immutable) and typically trigger a full application domain (or process) recycle to update. In order to externalize configuration away from a specific “hard-coded” API such as System.Configuration and integrate the Azure service configuration a number of helper methods are included in the ConfigurationHelper class, and used extensively throughout the application.

For example, to:

* Retrieve an integration configuration value
* Looking first in the Azure service configuration
* Next in web.config (or app.config)
* And finally substituting a default value

Your application would use the GetConfigValue<T>() helper as per:

|  |
| --- |
| ConfigurationHelper.GetConfigValue<int>("commandTimeout", timeout); |

## Logging

The limit of insight into an application’s health, behavior and performance is the effective limit of scale. While the default Azure diagnostics experience for .NET applications (System.Diagnostics + Windows Azure Diagnostics) is an excellent starting point, the default configuration does not work well in large-scale application. The CSFundamentals application includes a reference implementation for using NLog together with a scale-appropriate configuration of Windows Azure Diagnostics.

This approach consists of three core components:

* An NLog implementation of the generic ILogger interface (allowing other logging implementations such as log4net to be plugged in without rippling through the code base)
* The NLog configuration, shared amongst all of the deployed services, and updatable at runtime.
* The Windows Azure Diagnostics configuration, stored in diagnostics.wadcfg (one per deployed role).

The generic ILogger interface (defined in ILogger.cs) provides a generic interface for employing tracing in your application. It also has overloads for directly accepting Exception objects (removing the responsibility of correctly unrolling an exception from the developer) and a specific channel for tracing API calls.

Note: this additional TraceApi() overload is designed to streamline the experience of logging all external service or component calls with timing information and context.

Each level (Debug, Info, Warning, Error) has three overloads:

void Info(object message);

void Info(string fmt, params object[] vars);

void Info(Exception exception, string fmt, params object[] vars);

Allowing streamlined logging messages. The ILogger interface is used by requesting an ILogger from a logger factory (the default in this implementation is NLog) for a specific named channel, which can then be independently adjusted from the underlying log configuration.

As NLog does not easily support custom variables, Azure-specific information (role environment, local storage paths) are configured at run-time, using code from the NLogger::InitializeForCloud method. This extracts and configures the proper file paths, file names, and role context (such as role names). This enables publishing all logging files to the same blob container and maintaining unique file naming.

Note that we avoid the use of channel-per-class, preferring instead a logical association (Database, Caching, etc) for log channels. This is based on the assumption that any tracing that requires localizing a specific class and line number should be driven from a stack track contained in an exception.

The default NLog configuration incorporates four asynchronous targets (output files or sinks) and four named channels:

* **App**. The default application log target, stored in CSV format. Does not unroll exceptions to keep 1-line-per-log-event, and implements a rolling log appender (one file per hour, archive files are moved over to a separate archive directory).
* **Error**. The default exception target, stored in CSV format. Fully unrolls exceptions with stack traces and inner exceptions. Also implements a rolling log appender.
* **ApiTrace**. Separate file for logging component execution times and success/error codes, implementing a rolling log appender. Used for quantifying and diagnosing intra-service and intra-component call times, latency drift and variability.
* **Trace**.A pass through to Windows Azure Diagnostics table storage, using System.Diagnostics.Trace.

The four default named channels are ApiTrace (used to route to the ApiTrace target), Database, Cache and Scheduler. The default configuration may be easily extended to add additional named loggers.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  autoReload="false">  <!-- Register the custom extensions (Azure trace writer) -->  <extensions>  <add assembly="Microsoft.AzureCat.Patterns.Common"/>  </extensions>  <targets async="true">  <!-- Log standard (non-exception detail) messages to a rolling log file -->  <target name="App" xsi:type="File"  lineEnding="Default" autoFlush="true" keepFileOpen="false"  concurrentWrites="true" archiveEvery="Hour"  fileName="logs/ApplicationLog\_${date:format=YYYYmmDDHH}.txt"  archiveFileName="archive/ApplicationLog\_${date:format=YYYYmmDDHH}\_{#####}.log"  archiveNumbering="sequence"  maxArchiveFiles="720"  >  <layout xsi:type="CsvLayout">  <column name="time" layout="${longdate}" />  <column name="machinename" layout="${machinename}" />  <column name="role" layout="${machinename}" />  <column name="instance" layout="${machinename}" />  <column name="level" layout="${level}" />  <column name="logger" layout="${logger}" />  <column name="message" layout="${message}" />  <column name="exception" layout="${onexception:EXCEPTION OCCURRED\:${exception:format=type,message,method:maxInnerExceptionLevel=1:innerFormat=shortType,method}}" />  </layout>  </target>  <!--  File target used for detailed exception messages. Exception details and stack  trace are recorded in this target for ease of debugging.  -->  <target name="Error" xsi:type="FilteringWrapper"  condition="length('${exception}')>0">  <target name="Error\_File" xsi:type="File"  lineEnding="Default" autoFlush="false" keepFileOpen="false"  concurrentWrites="true" archiveEvery="Hour"  fileName="logs/ErrorLog.log"  archiveFileName="ErrorLog\_${date:format=YYYYmmDDHH}\_{#####}.log"  archiveNumbering="sequence"  maxArchiveFiles="720"  >  <layout xsi:type="CsvLayout">  <column name="time" layout="${longdate}" />  <column name="machinename" layout="${machinename}" />  <column name="role" layout="${machinename}" />  <column name="instance" layout="${machinename}" />  <column name="level" layout="${level}" />  <column name="logger" layout="${logger}" />  <column name="message" layout="${message}" />  <column name="exception" layout="${onexception:EXCEPTION OCCURRED\:${exception:format=type,message,method,stacktrace,tostring:maxInnerExceptionLevel=5:innerFormat=shortType,message,method}}" />  </layout>  </target>  </target>  <!-- Log API trace messages to a rolling log file -->  <target name="ApiTrace" xsi:type="File"  lineEnding="Default" autoFlush="true" keepFileOpen="false"  concurrentWrites="true" archiveEvery="Hour"  fileName="logs\ApiTraceLog.log"  archiveFileName="ApiTraceLog\_${date:format=YYYYmmDDHH}\_${time}\_{#####}.log"  archiveNumbering="sequence"  maxArchiveFiles="720"  >  <layout xsi:type="CsvLayout">  <column name="time" layout="${longdate}" />  <column name="machinename" layout="${machinename}" />  <column name="role" layout="${machinename}" />  <column name="instance" layout="${machinename}" />  <column name="api" layout="${event-context:item=api}" />  <column name="eventid" layout="${event-context:item=eventid}" />  <column name="action" layout="${event-context:item=action}" />  <column name="message" layout="${message}" />  </layout>  </target>  <!--  Write to the System.Diagnostics trace level monitored by Windows Azure Diagnostics.  These entries will route through to table storage.  -->  <target name="trace" xsi:type="FilteringWrapper" condition="level >= LogLevel.Warn">  <target name="azure\_trace" xsi:type="AzureEventLog"  layout="${message} ${onexception:EXCEPTION \:${exception:format=type,message,method,stacktrace:maxInnerExceptionLevel=5:innerFormat=shortType,message,method}}" />  </target>  </targets>  <!-- Rules; routing and matching -->  <rules>  <!-- Named configuration loggers -->  <logger name="ApiTrace" final="true" minlevel="Debug" maxlevel="Error" writeTo="ApiTrace" />  <logger name="Database" final="true" minlevel="Debug" maxlevel="Error" writeTo="App,Error,trace" />  <logger name="Cache" final="true" minlevel="Info" maxlevel="Error" writeTo="App,Error,trace" />  <logger name="Scheduler" final="true" minlevel="Debug" maxlevel="Error" writeTo="App,Error,trace" />  <!-- Write all remaining events to the default target -->  <logger name="\*" minlevel="Info" maxlevel="Error" final="true" writeTo="App,Error,trace" />  </rules>  </nlog> |

The diagnostics.wadcfg configuration file (for each Cloud Service role) provides an appropriate configuration for picking up and forwarding local telemetry data to a central storage account, including monitoring the archive directory for publishing custom logging files.

|  |
| --- |
| <!--  Diagnostics configuration for Web role management  -->  <DiagnosticMonitorConfiguration xmlns="http://schemas.microsoft.com/ServiceHosting/2010/10/DiagnosticsConfiguration" configurationChangePollInterval="PT5M" overallQuotaInMB="8192">  <DiagnosticInfrastructureLogs bufferQuotaInMB="256" scheduledTransferLogLevelFilter="Warning" scheduledTransferPeriod="PT5M" />  <Logs bufferQuotaInMB="512" scheduledTransferLogLevelFilter="Warning" scheduledTransferPeriod="PT5M" />  <Directories bufferQuotaInMB="1024" scheduledTransferPeriod="PT5M">  <!-- These three elements specify the special directories that are set up for the log types -->  <CrashDumps container="wad-crash-dumps" directoryQuotaInMB="100" />  <FailedRequestLogs container="wad-frq-logfiles" directoryQuotaInMB="100" />  <IISLogs container="wad-iis-logfiles" directoryQuotaInMB="256" />  <!--  Monitor the archive directory for local files (NLog, etc) that need to be  published to blob storage  -->  <DataSources>  <DirectoryConfiguration container="telemetry-logs" directoryQuotaInMB="4096">  <LocalResource name="LogStorage" relativePath="archive" />  </DirectoryConfiguration>  </DataSources>  </Directories>  <PerformanceCounters bufferQuotaInMB="512" scheduledTransferPeriod="PT5M">  <!-- The counter specifier is in the same format as the imperative diagnostics configuration API -->  <PerformanceCounterConfiguration counterSpecifier="\.NET CLR Memory(\_Global\_)\% Time in GC" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\.NET Data Provider for SqlServer(\*)\NumberOfReclaimedConnections" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Requests/Sec" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Requests Total" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Requests Succeeded" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Request Execution Time" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Requests In Application Queue" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Request Wait Time" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Requests Not Found" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET Apps v4.0.30319(\_\_Total\_\_)\Sessions Active" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\ASP.NET v4.0.30319\Requests Queued" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Memory\Pages/sec" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Memory\Committed Bytes" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Memory\Available MBytes" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Network Interface(\*)\Bytes Sent/sec" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Network Interface(\*)\Bytes Total/sec" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Network Interface(\*)\Output Queue Length" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\Processor(\_Total)\% Processor Time" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\System\Processor Queue Length" sampleRate="PT5M" />  <!-- Collect W3C thread pool usage -->  <PerformanceCounterConfiguration counterSpecifier="\W3SVC\_W3WP(\_Total)\Active Threads Count" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\W3SVC\_W3WP(\_Total)\Maximum Threads Count" sampleRate="PT5M" />  <PerformanceCounterConfiguration counterSpecifier="\W3SVC\_W3WP(\_Total)\Total Threads" sampleRate="PT5M" />  </PerformanceCounters>  <WindowsEventLog bufferQuotaInMB="256" scheduledTransferLogLevelFilter="Error" scheduledTransferPeriod="PT5M">  <!-- The event log name is in the same format as the imperative diagnostics configuration API -->  <DataSource name="Application!\*"/>  <DataSource name="System!\*"/>  <DataSource name="Windows Azure!\*" />  </WindowsEventLog>  </DiagnosticMonitorConfiguration> |

|  |
| --- |
| Note: this diagnostic configuration requires a total of 8GB of local storage. By default only 4 GB is allocated. In order to override this default your cloud service role configuration **MUST contain an entry for DiagnosticStore with a higher limit**.  The default location for logging data will be from the LogStorage local storage resource; **this MUST also be configured in your role configuration.** |

## Data Access

Working with a database-as-a-service (such as Windows Azure SQL Database) brings some additional challenges regarding developing an efficient and reliable data access layer. The code base includes data access wrappers for both single-database and sharded solutions, and demonstrates techniques for building and leveraging scale-out database designs.

The single-database wrapper methods in SqlAzureDalBase incorporate best practices for connection management and retry policies (note: this code base uses a retry policy implementation for Windows Azure SQL Database to demonstrate the technique using the Microsoft Enterprise Library Transient Fault Handling Application Block).

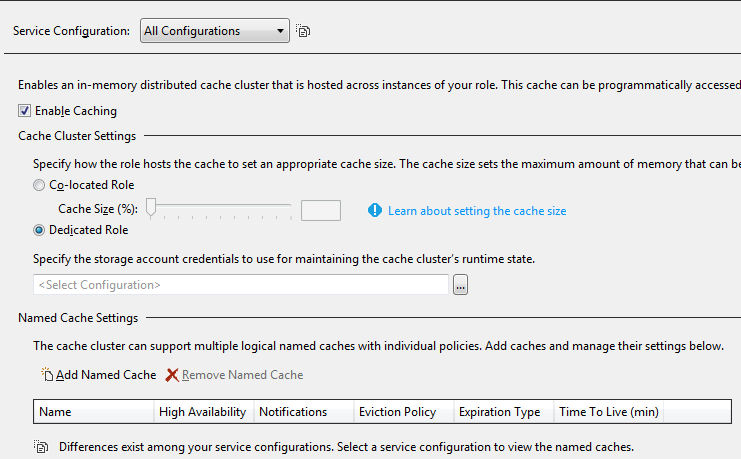
The data access layer also makes extensive use of Dapper, a micro-ORM for converting SQL calls and queries into .NET objects.

For example, here is an example of using Dapper to query for a specific value from a table without working directly with a SqlDataReader object:

|  |
| --- |
| var result = Query<DateTime?>(connectionString,  "SELECT MAX(eventdateUTC) FROM dbo.WADIISLogs",  commandType: CommandType.Text).FirstOrDefault(); |

## Caching

The CSFundamentals application makes extensive use of dedicated, role-based Azure Caching to streamline reads to commonly accessed information (such as user profile information). The caching service is deployed as a dedicated role within the CSFundamentals cloud service, using an abstraction layer over the Azure Caching client SDK.



## Scheduling

Scheduling and executing recurring work items is a very common aspect of modern cloud applications; this reference implementation leverages the Quartz.NET scheduler in a single-instance worker role to provide this functionality.

Note that this does not fulfill the Azure Compute SLA, and can incur downtime during role instance upgrade (as there are not multiple instances deployed across upgrade/fault domains).

The role is configured to retrieve a list of tasks from a remote blob container and schedule their execution.

# Application Request Routing (ARR)

In order to seamlessly transfer and affinitize traffic between multiple cloud services the CSFundamentals application uses the [Application Request Routing](http://www.iis.net/downloads/microsoft/application-request-routing) functionality in IIS.



Affinity through the router service enables resources to be associated with a set of users, increasing locality and minimizing cross-talk between services (essentially reducing the number of required database connections and improving overall performance).

The ARR routing is configured through the installing the ARR module in a startup task, registering custom extensions and configuring the specific rules via web.config. The startup task installs the custom routing extension, and configures ARR via appcmd:

|  |
| --- |
| %~dp0gacutil /i Microsoft.AzureCat.Router.LoadBalanceThenCustomCookieProvider.dll >> installlog.txt 2>&1  md "%~dp0appdata"  reg add "hku\.default\software\microsoft\windows\currentversion\explorer\user shell folders" /v "Local AppData" /t REG\_EXPAND\_SZ /d "%~dp0appdata" /f  %~dp0webpicmd /Install /AcceptEula /SuppressReboot /Products:ARRv2\_5 >> installlog.txt 2>&1  reg add "hku\.default\software\microsoft\windows\currentversion\explorer\user shell folders" /v "Local AppData" /t REG\_EXPAND\_SZ /d %%USERPROFILE%%\AppData\Local /f >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/proxy /enabled:"True" /commit:apphost >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/proxy /reverseRewriteHostInResponseHeaders:"False" /commit:apphost >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/rewrite/allowedServerVariables /+[name='HTTP\_ACCEPT\_ENCODING'] /commit:apphost >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/rewrite/allowedServerVariables /+[name='HTTP\_COOKIE'] /commit:apphost >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/rewrite/allowedServerVariables /+[name='URL'] /commit:apphost >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/rewrite/allowedServerVariables /+[name='ALL\_HTTP'] /commit:apphost >> installlog.txt 2>&1  %windir%\system32\inetsrv\appcmd.exe set config -section:system.webServer/rewrite/allowedServerVariables /+[name='ALL\_RAW'] /commit:apphost >> installlog.txt 2>&1  certutil -f -p csfundamentals\_certSSLpwd -importpfx csfundamentals\_certSSLname.pfx >> installlog.txt 2>&1 |

The rules, defined in web.config set up the following flow, where a pod is defined as deployment of the CSFundamentals application:

* If user has a registered cookie userpod, then route the user to designated pod (URL)
* If the user has no registered cookie userpod, then route to user to a random pod

# Application Scenario Implementation

The CSFundamentals application implements a custom membership provider implementation against a scaled-out relational store (based on Windows Azure SQL Database), with registration and login support, as well as the ability to publish and retrieve comments.

## Database Design

Much of the reference implementation is related to working against a scaled-out relational data store based on Windows Azure SQL Database. This section explores the design, partitioning model and scale-out aware data access layer implemented by the CSFundamentals application.

The database is designed as a two-tier hybrid partitioning model. Users have two external unique identifiers – username and email address. These in turn map to an internal user ID (a GUID or uniqueidentifier). There are two sets of databases:

* **Root** databases store the mapping information between usernames, email and user IDs.
* **UserProfile** databases store the actual user content (profiles, posts, etc).

Both of these tiers are partitioned and scaled-out. The root databases use a hashing technique to map the username (or email) into a shard identifier, while the userprofile databases depend on database ranges stored in a configuration map.

Many shard schemes leverage a single central metadata database to store the shard map; under high loads this single database becomes a critical bottleneck and failure point. To avoid this, the root data store is scaled out across a small number of databases using a deterministic partitioning scheme.



When registering a new user (by name or email address), the first step in the application code is to run the creation stored procedures against the root databases. The steps involved are:

1. Hash the input string (user name or email address); the CSFundamentals application uses the default .NET GetHashCode as a placeholder. One should use a stable hashing algorithm such as the [MurMur3](http://en.wikipedia.org/wiki/MurmurHash) hash or [CityHash](http://en.wikipedia.org/wiki/CityHash) implementation.
2. Bucket the hash value based on the number of configured root databases (hash value MOD number of databases).
3. Look the requisite connection string up against the list of configured root databases.
4. Execute the requisite stored procedures.



The actual data content uses a range-based partitioning scheme (the partitioning information is stored in a blob-accessible configuration file which is downloaded when the application starts up) to split the full range of possible values into a series of contiguous segments. To access user information (profile, content, etc) the code needs to:

1. Hash the input string (user name or email address).
2. Map the hash value against the appropriate shard range (by looking at the list of shards)
3. Retrieving the appropriate connection string from the shard range configuration
4. Execute the requisite stored procedures or queries



The UserDataAccess class provides the application specific logic on top of the individual data access functions in SampleShardedDalBase (which encapsulates the root and user database shard mapping schemes) and ShardedDalBase (which provides abstractions for working against scale-out data sets).

## CSFundamentals Shard Map

The CSFundamentals application’s data access layer support many Root databases and many UserProfile databases with the shard map contained in the file “RootUsersShardMap.xml”. The xml below shows 2 Root databases and 2 UserProfile databases.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <LogicalShards>  <Shard LogicalRootName="RootDB" ShardID="0"  ServerName="tcp:csfundamentals\_dbservername.database.windows.net"  AdminUser="csfundamentals\_dbusername@csfundamentals\_dbservername"  AdminPassword="csfundamentals\_dbpassword"  DatabaseName="CSFundamentalsRootDB\_000"  ShardColumnType="int"  RangeMin="0" RangeMax="1" />  <Shard LogicalRootName="RootDB" ShardID="1"  ServerName="tcp:csfundamentals\_dbservername.database.windows.net"  AdminUser="csfundamentals\_dbusername@csfundamentals\_dbservername"  AdminPassword="csfundamentals\_dbpassword"  DatabaseName="CSFundamentalsRootDB\_001"  ShardColumnType="int"  RangeMin="1" RangeMax="2" />  <Shard LogicalRootName="UserProfile" ShardID="0"  ServerName="tcp:csfundamentals\_dbservername.database.windows.net"  AdminUser="csfundamentals\_dbusername@csfundamentals\_dbservername"  AdminPassword="csfundamentals\_dbpassword"  DatabaseName="CSFundamentalsUserProfileDB\_000"  ShardColumnType="int"  RangeMin="-2147483648" RangeMax="-1" />  <Shard LogicalRootName="UserProfile" ShardID="1"  ServerName="tcp:csfundamentals\_dbservername.database.windows.net"  AdminUser="csfundamentals\_dbusername@csfundamentals\_dbservername"  AdminPassword="csfundamentals\_dbpassword"  DatabaseName="CSFundamentalsUserProfileDB\_001"  ShardColumnType="int"  RangeMin="-1" RangeMax="2147483647" />  </LogicalShards> |

Both sets of shards, RootDB and UserProfile, have a min and max range where the min (left) is inclusive and the right (max) is exclusive except where the max is Int.Max value ending the range making this the only right inclusive range. The pattern to follow for modifying the “RootUsersShardMap.xml” file is to give each database a ShardID starting at 0 and progressive, contiguous min/max range for each database shard.

## User Registration

User registration is handled through customizing the standard MVC4 AccountController class (in the web role, under Controllers). In the Register method, we accept the RegisterModel (the standard user account information), and use the scale-out database aware user information data access layer component (UserDal) to register the user.

// Attempt to register the user

var userInfo = UserDal.RegisterUser(

model.UserName, model.Password, model.UserEmail,

model.UserFirstName, model.UserLastName, model.UserNickName);

This bypasses the typical CustomWebSecurity class, which provides database agnostic interfaces for generating and manipulating the account table information. The user dal encapsulates the management of cache and relational data, as well as creating the requisite data in the appropriate sequence.

|  |
| --- |
| // Hash and valdiate the password complexity  string hashedPassword = Crypto.HashPassword(password);  if (hashedPassword.Length > 128)  throw new ArgumentException("An invalid password was provided");  // Generate a new user ID that will be used as an internal key  Guid userId = System.Guid.NewGuid();  int shardId = shardMeta.CalcProfileUserIdShardKey(userId);  // UserName and Email are stored as separate entry points in the root  // mapping  CreateUserInRoot(userId, shardId, userName);  if (email != null)  CreateEmailInRoot(userId, shardId, email);  // Create the appropriate record in the user data database  CreateUserDataRecord(userName, email, firstName, lastName,  nickName, hashedPassword, userId, shardId);  // Generate the new user object  CSFundamentalsUser user = new CSFundamentalsUser(userName, firstName, lastName,  email, nickName, shardId, userId);  return user; |

## User Login

User login is also handled through customization of the standard MVC4 AccountController class (in the web role, under Controllers). In the Login method, we accept a username and password, and validate them against the backing store.

|  |
| --- |
| // Query the root database to execute dbo.GetUserIDShardID  var result = RootShardQuery(userName, "dbo.GetUserIDShardID",  new { UserName = userName, Email = String.Empty },  commandType: CommandType.StoredProcedure,  methodName: "GetUserIDShardID").FirstOrDefault();  var userId = (Guid)result.UserID;  var shardId = (int)result.UserIDShardID;  // Query the user content database for the password and profile information  var hashedPassword = UserShardQuery<string>(userId,  "SELECT [Password] FROM dbo.AuthUserPwd WHERE UserID = @UserId",  new { UserId = userId }, methodName: "GetPassword").FirstOrDefault();  var isValid = Crypto.VerifyHashedPassword(hashedPassword, password);    if (!isValid)  return null;  //Now that u have a userID, shardid and SQLConn, validate the password & load the user  var userInfo = UserShardQuery(userId,  "SELECT UserName, FName, LName, PrimaryEmail, Avatar FROM dbo.UserBase WHERE UserID = @UserId",  new { UserId = userId }, methodName: "GetUserInfo").FirstOrDefault();  var user = new CSFundamentalsUser(  userInfo.UserName,  userInfo.FName,  userInfo.LName,  userInfo.PrimaryEmail,  userInfo.Avatar,  shardId,  userId);  // Publish the user information to cache  cacheClient.Put<CSFundamentalsUser>(userId.ToString(), user); |

If the password is valid we populate the cache with the user profile, such that it may be efficiently accessed on subsequent calls.

## Add Comment

Adding a new comment publishes the data into the shared cache, then inserts into the backing relational store through a stored procedure. There are potential optimizations in this code path, including making the database call asynchronous (trading throughput for response latency).

|  |
| --- |
| public bool AddComment(CSFundamentalsUser user, String comment)  {  bool ret = false;    // try accessing the cache and updating comments  try  {  // Generate the new post object  Post commentPost = new Post()  {  PostContent = comment,  URL = "TBD",  };  commentPost.Tags.Add("comment");  user.Profile.MyData.Comments.Add(commentPost);    // Check the cache to see if posts are present  string key = user.UserId + "\_Comment";  var userPosts = cacheClient.Get<List<Post>>(key);    // On a cache miss - pull comments from database  if (userPosts == null)  {  var pageSize = ConfigurationHelper.GetConfigValue<int>("CommentPageSize", 5);  var postData = UserShardQuery<PostData>(user.UserId, "dbo.GetPostings",  new { UserID = user.UserId, PageSize = pageSize, OlderThan = DateTime.UtcNow },  commandType: CommandType.StoredProcedure,  methodName: "GetPostings");    userPosts = postData.Select(e => new Post()  {  PostContent = e.PostContent,  Tags = new List<string>(),  URL = String.Empty,  // TODO: add a field for created time  }).ToList();  }  // Update cache - todo put a bounded size on the number of comments in cache  userPosts.Add(commentPost);  cacheClient.Put<List<Post>>(key, userPosts);  // Update the database  UserShardExecute(user.ShardID, (connection) =>  {  connection.Execute("dbo.AddPost",  new { UserID = user.UserId, PostContent = comment, DateCreated = DateTime.UtcNow },  commandType: CommandType.StoredProcedure);  }, "AddPost");    ret = true;  }  catch (Exception excp)  {  Logger.Error(excp, "Could not add comment in database. Userid={0} ShardId={1} PODInfo={2}",  user.UserId, user.ShardID, user.UserPODInfo);  }  return ret;  } |

Note the use of the UserShardExecute overload from the shard-aware data access layer, which maps the user’s shard ID into database connection to execute the AddPost stored procedure.

## Get Comments

Retrieving comments for display is a mirror of AddComment, looking first in the distributed cache for the comment content, then retrieving from the database and updating the cache if/as necessary.

|  |
| --- |
| public IEnumerable<Post> GetComment(CSFundamentalsUser user)  {  var pageSize = ConfigurationHelper.GetConfigValue<int>("CommentPageSize", 5);  IEnumerable<Post> res = Enumerable.Empty<Post>();  try  {  // Grab the posts from the cache  string key = user.UserId + "\_Comment";  res = cacheClient.Get<List<Post>>(key);  if (res != null)  {  user.Profile.MyData.Comments = res.ToList();  }  else  {  // If not available in the cache pull from the database  // and populate the cache  var postData = UserShardQuery<PostData>(user.UserId, "dbo.GetPostings",  new { UserID = user.UserId, PageSize = pageSize, OlderThan = DateTime.UtcNow },  commandType: CommandType.StoredProcedure,  methodName: "GetPostings");  res = postData.Select(e => new Post()  {  PostContent = e.PostContent,  Tags = new List<string>(),  URL = String.Empty,  // TODO: add created time  });  // Add to the cache  cacheClient.Put<List<Post>>(key, res.ToList());  }  }  catch (Exception excp)  {  Logger.Error(excp, "Could not get comments off Cache");  }  return res;  } |

# Telemetry Monitor Scenario Implementation

The CSFundamentals application implements a task-driven telemetry collection and transformation service, which aggregates data from multiple sources (Windows Azure Diagnostics, Windows Azure SQL Database, etc.) into a consolidated destination database. The scheduling and execution of these tasks is driven by Quartz.NET running in a worker role.



The single-instance worker role spins up a Quartz.NET scheduler with a polling job. The responsibility of the polling job is to monitor an external blob container for the job configuration file. This file contains the telemetry collection jobs to be executed by the scheduler service, and allows the job configuration to be updated at run-time without redeploying the service.

The set of jobs which ship with the code sample (QuartzJobs.xml) handle the collection, transformation and importation of telemetry data from multiple sources into a consolidated operations database.



## Task – Consolidate WAD Performance Counters

The WAD performance counter task is responsible for retrieving and consolidating performance counter data from the WADPerformanceCounters table. While table storage provides an excellent scalable store for diagnostic information, it is not as well suited for aggregations and calculations.

|  |
| --- |
| <job>  <name>PerfLogsImportJob</name>  <group>Telemetry</group>  <description>Import Perf Mon data from Table Storage</description>  <job-type>Microsoft.AzureCat.Patterns.Tasks.SqlAzureMgmt.PerfMonImportTask, Microsoft.AzureCat.Patterns.Tasks.SqlAzureMgmt</job-type>  <durable>true</durable>  <recover>false</recover>  <job-data-map>  <entry>  <key>CloudStorageAccount</key>  <value>csfundamentals\_storageaccountname</value>  </entry>  <entry>  <key>PrivateKey</key>  <value>[my private key]</value>  </entry>  <entry>  <key>SQLAzureConnString</key>  <value>Data Source=dbname.database.windows.net;Initial Catalog=OpsStatsDb;User ID=username@database;Password=[password];Connect Timeout=30;Encrypt=False;TrustServerCertificate=False</value>  </entry>  </job-data-map>  </job> |

The task runs in four steps:

1. Retrieve the task configuration settings from the task configuration context (as set by the task configuration XML file)
2. Retrieve the latest timestamp of data that has been loaded into the operations database
3. Retrieve all performance counter data between the last timestamp and T-1 minute.
4. Bulk load this data into the operations database

|  |
| --- |
| public virtual void Execute(IJobExecutionContext context)  {  // Job configuration here  var connString = context.JobDetail.JobDataMap.GetString("SQLAzureConnString");  var accountName = context.JobDetail.JobDataMap.GetString("CloudStorageAccount");  var privateKey = context.JobDetail.JobDataMap.GetString("PrivateKey");  Execute(() =>  {  try  {  Logger.Info("PerfMonImportTask.Execute: started at {0}",  DateTime.UtcNow.ToShortDateString());  var query = new PerfMonImportHelper(accountName, privateKey);  var startDate = context.PreviousFireTimeUtc != null ?  context.PreviousFireTimeUtc.Value.DateTime.AddMinutes(-1) :  DateTime.UtcNow.AddMinutes(-10);  var endDate = DateTime.UtcNow.AddMinutes(-1);  var results = query.QueryTraceData(2, startDate, endDate);  BulkInsert(connString, results);  Logger.Info("PerfMonImportTask.Execute: {0} - Time Interval: {2} - {3}",  results.Count, startDate, endDate);  }  catch (Exception ex0)  {  Logger.Error(ex0, "[Telemetry] PerfMonImportTask.Execute: Failed! {0}", ex0.Message);  }  }, "PerfMonImportTask");  } |

## Task – Gather SQL Dynamic Management Views

The SQLDMV task is responsible for retrieving and consolidating dynamic management view data from the set of configured databases. This task leverages parallel scatter/gather queries to simultaneously gather data from multiple disparate databases.

The task runs in four steps:

1. Retrieve the task configuration settings from the task configuration context (as set by the task configuration XML file), along with the list of shards (databases)
2. Import the database size DMVs from the target databases, and bulk load into the operations database
3. Import the request stats DMVs from the target databases, and bulk load into the operations database
4. Import the query stats DMVs from the target databases, and bulk load into the operations database

Each of the import tasks uses the ExecuteFanOutSelect method from the FanOutQueryBase class to perform parallel select queries against the destination databases.

|  |
| --- |
| private void ImportDBSize(string connString, IList<IShard> listShard)  {  var query = new FanOutQueryBase("Adhoc");  query.ShardList = listShard;  DataTable res = query.ExecuteFanOutSelect(GetQueryDBSize());  DataTable dtNew = new DataTable();  dtNew.Columns.Add("timestampKey", typeof(Int64));  dtNew.Columns.Add("eventdateUTC", typeof(DateTime));  dtNew.Columns.Add("db\_name", typeof(string));  dtNew.Columns.Add("SKU", typeof(string));  dtNew.Columns.Add("MAXSizeGB", typeof(double));  dtNew.Columns.Add("DBPcntFull", typeof(double));  dtNew.Columns.Add("SizeInGB", typeof(double));  foreach (DataRow dr in res.Rows)  {  dtNew.Rows.Add(dr["timestampKey"],  dr["eventdateUTC"],  dr["db\_name"],  dr["SKU"],  dr["MaxSizeGB"],  dr["DBPcntFull"],  dr["SizeInGB"]  );  }  BulkInsert(connString, dtNew, "DatabaseSize");  } |

# Configuration and Deployment

This section outlines the steps to configure and deploy the CSFundamentals solution and its services. A description of each step is described for configuration and deployment here:

1. Get an Azure subscription and create services for the CSFundamentals services: Hosted Services in Windows Azure, Azure SQL Database server and databases, Windows Azure Storage account
2. Download the CSFundamentalsSoicial solution and C# source code
3. Create Certificates - management certificates, certificate for Remote Desktop usage and creating SSL certificate (the SSL certificate will be added to the RouterService project)
4. Change the placeholder “csfundamentals\_” settings in the appropriate project files (see below) to ***your*** Windows Azure service values including modifying the QuartzJobs.xml and the RootUsersShardMap.xml
5. Package and deploy the 3 hosted services:
   1. CSFundamentals in the CSFundamentals Visual Studio solution
   2. SchedulerService in the CSFundamentals Visual Studio solution
   3. CustomCookieBasedRouter in the CustomCookieBasedRouter Visual Studio solution (RouterService)

## Prerequisites

When configuring the CSFundamentals services, there are prerequisites to configuration and deployment. These are listed below followed by the steps for configuration, packaging and deployment to Azure.

* Valid Azure Subscription and available quota for the following services:
  + Hosted Services (3 minimum):
    - RouterService: 1
    - CSFundamentals: at least 2 (depends on how many “Pods” are deployed)
    - Note: if only 1 CSFundamentalsService Hosted Service is deployed then only 1 Hosted Service is need and the RouterService will not need to be deployed since its purpose is to route traffic to 2 or more CSFundamentalsServices).
    - SchedulerService: 1
    - Note: All Hosted Services instances are configured to a VM size of Small requiring 1 core per instance (3 cores minimum).
  + Storage Accounts (1):
    - Diagnostics: WAD for the hosted services
    - SchedulerService: Quartz job information
    - CSFundamentals: Root and UserProfile database connection string information in the RootUsersShardMap.xml
  + Windows Azure SQL Database Server (1 SQL Database server)
  + SQL Database (5 databases):
    - SchedulerService database: 1 database to store the OpsStatsDB collected data
    - CSFundamentals sharded databases:
      * 2 or more Root databases
      * 2 or more UserProfile databases
  + Certificates (1 minimum, up to 3 certificates):
    - Azure Management certificate: create a management certificate that is uploaded to Azure portal “Management Certificates”. The certificate is necessary to use Powershell scripts to manage the subscription and to deploy the services. This certificate (.pfx) must also be imported to your “My” certificate store. To verify, run certmgr.msc and go to (Current User)\Personal\Certificates to local the certificate.
    - SSL certificate: the CSFundamentalsSoical uses SSL and requires the certificate be uploaded to the Hosted Services 🡪 Certificates location. The certificate thumbprint is to be included in the configuration files when deployed.
    - Remote Desktop (RDP) certificate: it’s recommended to enable remote desktop protocol (RDP) for each Hosted Service deployed. The RDP certificate created will be uploaded to the same location as the SSL certificate. RDP will be manual enabled on each Hosted Service from Visual Studio when the deployment package is created. The RDP certificate must be specified, the rdp user name and the rdp password. The user name and password can also be changed after the Hosted Service is deployed.

In the CSFundamentals Visual Studio solution, these values are needed from your Windows Azure prerequisites. Use the table below as a checklist for your Azure subscription values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Description** | **CSFundamentals Solution Tags** | **Your values here** |
| Subscription | Subscription Id | csfundamentals\_subscriptionid |  |
|  | Subscription Name | csfundamentals\_subscriptionname |  |
| Router/Pod | Hosted Services | csfundamentals\_dnsname |  |
|  | Pod Service Count | csfundamentals\_podcount |  |
| Scheduler Service | Hosted Services | csfundamentalsscheduler\_dnsname |  |
| Storage | Storage Account Name | csfundamentals\_storageaccountname |  |
|  | Storage Account Key | csfundamentals\_storageaccountkey |  |
|  | Storage Container | csfundamentals\_storagecontainer |  |
| Certificates | Mgmt Certificate Name | csfundamentals\_certMGMTname |  |
|  | Mgmt Certificate Password | csfundamentals\_certMGMTpwd |  |
|  | SSL Certificate Name | csfundamentals\_certSSLname |  |
|  | SSL Certificate Password | csfundamentals\_certSSLpwd |  |
|  | SSL Certificate Thumbprint | csfundamentals\_certSSLthumbprint |  |
|  | RDP Certificate Name | csfundamentals\_certRDPname |  |
|  | RDP Certificate Password | csfundamentals\_certRDPpwd |  |
|  | RDP Certificate Thumbprint | csfundamentals\_certRDPthumbprint |  |
| rdpuser | RDP User | csfundamentals\_RDPuser |  |
|  | RDP Password | csfundamentals\_RDPpwd |  |
| Database | DB Server | csfundamentals\_dbservername |  |
|  | DB UID | csfundamentals\_dbusername |  |
|  | DB PWD | csfundamentals\_dbpassword |  |
|  | Root DB Count | csfundamentalsrootdbcount |  |
|  | Root DB Name (prefix) | CSFundamentalsRootDB |  |
|  | UserProfile DB Count | csfundamentalsUserProfledbcount |  |
|  | UserProfile DB Name (prefix) | CSFundamentalsUserProfileDB |  |
| Azure Cache | Worker Role Name | CSFundamentalsCache.WorkerRole |  |
| Datacenter | WA DC Name | csfundamentals\_walocation |  |

Note: there are 2 methods described for configuring and deploying to Windows Azure in these sections:

1. [Configuration and Deployment from Visual Studio](#_Configuration_and_Deployment_2)
2. [Configuration and Deployment from Powershell](#_Configuration_and_Deployment_1)

## Configuration and Deployment from Visual Studio

This sections details the steps to configuring and deploying from Visual Studio, requiring the project files to be changed manually and also creating the Windows Azure services in the Windows Azure management portal.

Start by creating the Windows Azure Hosted Services, the databases in SQL Database databases and the Windows Azure Storage container following these steps:

1. Create the Hosted Services

From Visual Studio, right click on each service and click “Package” to create the Hosted Service packages. The 3 services created:

* RouterService service: use the base csfundamentals\_dnsname for this Hosted Service. This service will route user connections to the CSFundamentalsService Hosted Services which will use the dnsname plus an incremental integer. For example, csfundamentals\_dnsname1, csfundamentals\_dnsname2, etc. for each CSFundamentalsService Hosted Service (aka pod) deployment.
* CSFundamentalsService service: this service is packaged only once but deployed multiple times for each pod specified in the CSFundamentalsService podCount from the service configuration (ServiceConfiguration.Cloud.cscfg). For each CSFundamentalsService Hosted Service deployed to Windows Azure, give it a unique name starting with csfundamentals\_dnsname1 then csfundamentals\_dnsname2 and so on.
* SchedulerService: package and deploy the service to Windows Azure.

1. Create the OpStatsDB in the WA SQL Database

Create the OpStatsDB SQL Database by creating the database in the WA SQL Database Portal. Once the database is created, run the T-SQL scripts to create database objects found in the CSFundamentals\Data\Microsoft.AzureCat.Patterns.Data.SqlAzure.OpsStatsDB\dbo directory in the following order: Tables, Functions and Views.

1. Create the RootDB and UserProfileDB sharded databases in the WA SQL Database

Depending on the number of sharded databases required by your application, create 2 or more of each database: CSFundamentalsRootDB\_001, CSFundamentalsRootDB\_002 and so on. Create the UserProfileDBs in a similar fashion: CSFundamentalsUserProfileDB\_001, CSFundamentalsUserProfileDB\_002, ... Once the databases are created, run the T-SQL scripts found in the CSFundamentals\Data\Microsoft.AzureCat.Patterns.Data.SqlAzure.OpsStatsDB\dbo directory in the following order: Tables and Stored Procedures.

1. Windows Azure Storage Container

Create the storage container name in the Windows Azure blog storage. Later, the Config\QuartzJobs.xml and Config\RootUsersShardMap.xml files will be uploaded to this storage container.

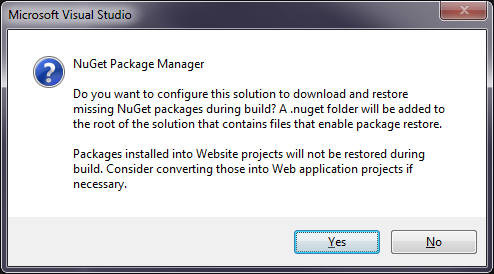
## Configuration Files

With the list of Azure subscription values specific for your CSFundamentals deployment, this section will walkthrough how to configure each of the project file necessary for deployment. Use the table above to list your values and change the Solution Tags in the CSFundamentalsSoical solution projects. The table below lists the files to be modified:

|  |  |
| --- | --- |
| **Projects** | **Files to modify** |
| CSFundamentalsService | ServiceConfiguration.Cloud.cscfg |
| CSFundamentalsService.WebRole | Web.config |
| CSFundamentalsService.WebRole | Config\RootUsersShardMap.xml |
| RouterService | ServiceConfiguration.Cloud.cscfg |
| RouterService.WebRole | Web.config |
| RouterService.WebRole | startup.cmd |
| RouterService.WebRole | CSFundamentals.Router.WebRole.csproj |
| SchedulerService | ServiceConfiguration.Cloud.cscfg |
| SchedulerService.WorkerRole | Config\QuartzJobs.xml |

### Build the CSFundamentals Solution

To build the solution, open the CloudServiceFundamentals.sln file and select the “Release” configuration to BUILD. Enable NuGet packages to be automatically downloaded from the internet (NuGet.org) by righting click on “Solution ‘CloudServiceFundamentals” at the top of the Solution Explorer view. Click on “Enable NuGet Package Restore” menu item and click Yes on the Visual Studio dialog to confirm, as show below.



The values starting with “csfundamentals\_” are to be replaced in each of these files.

### CSFundamentalsService – ServiceConfiguration.Cloud.cscfg

* **csfundamentals\_dnsname**

httpRedirectHome and httpsRedirectLogin values that reference the RouterService

* **csfundamentals\_podcount**

podCount *value* which is the number of CSFundamentalsService hosted services which is the number of pods deployed

* **csfundamentals\_storageaccountname**

WA storage account name, modify Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString and Microsoft.WindowsAzure.Plugins.Caching.ConfigStoreConnectionString

* **csfundamentals\_storageaccountkey**

WA storage account primary key, modify Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString and Microsoft.WindowsAzure.Plugins.Caching.ConfigStoreConnectionString

* **SSL Certificate Thumbprint**

Replace ABABABABABABABABABABABABABABABABABABABAB with the thumbprint of your SSL certificate. This certificate must be uploaded to your hosted services before deployment. To view the Thumbprints in your local store (my local machine), execute: “gci.exe cert:\LocalMachine\My” from the command-line. The certificate with subject CN=\*.cloudap.net is the SSL cert.

* **RDP Certificate Thumbprint**

Replace CDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD with the thumbprint of your RDP certificate. This certificate must be uploaded to your hosted services before deployment. To view the Thumbprints in your local store (my local machine), execute: “gci.exe cert:\LocalMachine\My” from the command-line.

### CSFundamentalsService.WebRole – Web.config

* **csfundamentals\_storageaccountname**

WA storage account name, modify StorageAccountShardMap

* **csfundamentals\_storageaccountkey**

WA storage account primary key, modify StorageAccountShardMap

* **csfundamentals\_storagecontainer**

WA Blob Storage container name, modify ContainerNameShardMap

### CSFundamentalsService.WebRole – Config\RootUsersShardMap.xml

* **csfundamentals\_dbservername**

Modify the ServerName with the name of the SQL Database server

* **csfundamentals\_dbusername**

Modify the AdminUser with the user id and the name of the SQL Database server

* **csfundamentals\_dbpassword**

Modify the AdminPassword with the password

* **CSFundamentalsRootDB**

Modify the DatabaseName for each Root database (each database must have a unique name)

* **CSFundamentalsUserProfileDB**

Modify the DatabaseName for each UserProfile database (each database must have a unique name)

### RouterService – ServiceConfiguration.Cloud.cscfg

* **csfundamentals\_storageaccountname**

WA storage account name, modify Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString

* **csfundamentals\_storageaccountkey**

WA storage account primary key, modify Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString

* **SSL Certificate Thumbprint**

Replace ABABABABABABABABABABABABABABABABABABABAB with the thumbprint of your SSL certificate. This certificate must be uploaded to your hosted services before deployment. To view the Thumbprints in your local store (my local machine), execute: “gci.exe cert:\LocalMachine\My” from the command-line. The certificate with subject CN=\*.cloudap.net is the SSL cert.

* **RDP Certificate Thumbprint**

Replace CDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD with the thumbprint of your RDP certificate. This certificate must be uploaded to your hosted services before deployment. To view the Thumbprints in your local store (my local machine), execute: “gci.exe cert:\LocalMachine\My” from the command-line.

### RouterService.WebRole – Web.config

* **csfundamentals\_dnsname**

httpRedirectHome and httpsRedirectLogin values that reference the RouterService service

### RouterService.WebRole – startup.cmd

* **csfundamentals\_certSSLname**

SSL certificate name in the format “csfundamentals\_certSSLname.pfx”

* **csfundamentals\_certSSLpwd**

Replace with the password for the SSL certificate

### RouterService.WebRole – CSFundamentals.Router.WebRole.csproj

* **csfundamentals\_certSSLname** (csfundamentals\_certSSLname.pfx)

Update the SSL certificate, in the CSFundamentals.Router.WebRole project:

* Delete the csfundamentals\_certSSLname.pfx from the project
* Add Existing item: find your SSL .PFX file and add it to the project
* After the .PFX file is added, right click the .PFX file and select properties
* For “Copy to Output Directory” select “Copy always”

### SchedulerService – ServiceConfiguration.Cloud.cscfg

* **csfundamentals\_storageaccountname**

WA storage account name, modify Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString

* **csfundamentals\_storageaccountkey**

WA storage account primary key, modify Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString

* **RDP Certificate Thumbprint**

Replace CDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD with the thumbprint of your RDP certificate. This certificate must be uploaded to your hosted services before deployment. To view the Thumbprints in your local store (my local machine), execute: “gci.exe cert:\LocalMachine\My” from the command-line.

### SchedulerService.WorkerRole – Config\QuartzJobs.xml

* **csfundamentals\_storageaccountname**

WA storage account name, modify CloudStorageAccount and StorageAccount

* **csfundamentals\_storageaccountkey**

WA storage account primary key, modify PrivateKey and StorageAccount

* **csfundamentals\_storagecontainer**

WA Blob Storage container name, modify ContainerName

* **csfundamentals\_dbservername**

Modify the Data Source and User ID with the name of the SQL Database server

* **csfundamentals\_dbusername**

Modify the User ID with the user id and the name of the SQL Database server

* **csfundamentals\_dbpassword**

Modify the Password

Note: For the OpsStatsDB database specified in the QuartzJobs.xml, the same SQL Database server, user name and password are used as for the RootDB and UserProfileDB sharded databases.

### Deploying the Solution to Windows Azure with Visual Studio

These steps assume the Azure Subscription has already been created and the necessary services have been provisioned as described in the previous section on configuration.

### Upload Management Certificate

The management certificate for your Azure subscription needs to be uploaded to the Windows Azure Portal – Management Certificates. For more information on Certificates in Windows Azure [click on this link](http://msdn.microsoft.com/en-us/library/windowsazure/gg981935.aspx).

### Database Configuration: Modify the XML Configuration File

Modify the QuartzJobs.xml and RootUsersShardMap.xml files to reflect the SQL Database servers, database names, user ids, passwords and sharding ranges.

### Publish the 3 services from Visual Studio: CSFundamentalsService, RouterService and SchedulerService

* Publish the Package for the CSFundamentalsService
  + Right click CSFundamentalsService and click Package
  + Enable Remote Desktop for the package by checking the checkbox and click “Settings.”
  + Verify that the RDP certificate is selected.
  + Right click CSFundamentalsService and click Publish
    - Publish the CSFundamentalsService for each “Pod” to be deployed
    - The service name will be changed for each deployment “csfundamentals\_dnsnameXXX” where XXX starts at 1 and is incremented for each service
* Publish the Package for the RouterService
  + Right click RouterService and click Package
  + Enable Remote Desktop for the package by checking the checkbox and click “Settings.”
  + Verify that the RDP certificate is selected.
  + Right click RouterService and click Publish
* Publish the Package for the SchedulerService
  + Right click SchedulerService and click Package
  + Enable Remote Desktop for the package by checking the checkbox and click “Settings.”
  + Verify that the RDP certificate is selected.
  + Right click SchedulerService and click Publish

## Configuration and Deployment with Powershell

This section describes the Powershell scripts provided to configure the CSFundamentals solution for building and deploying to Azure. The scripts are organized and numbered by the order they should be executed along with a description of the steps. There are manual steps involved too that must be followed in the order described which include uploading the management certificate to the Azure portal, modifying the RoutingService project (selecting your SSL certificate) and creating the deployment packages for each service.

Find the Powershell script directory located under the “PowerShellDeployment” directory and follow these steps executing the numbered scripts in the number directories.

### 0-DefaultAzureValues

* **\*\* Manual step:** Modify the file AzureValue.csv. See Appendix B: Azure Subscription Configuration Settings for a description. Add your values to the right of the commas.
* 1-GenerateDefaults.ps1 – Using the AzureValue.csv file, generate the SetDefaults.ps1 file used by other CSFundamentals Powershell scripts.

### 1-Certificates

* 1-CreateCertificates.ps1 – Create the certificates for SSL, RDP and Azure Management then load them into the local, relevant certificate stores. Generate the CertThumprints.csv file containing the thumbprints for the SSL and RDP certificates to be used in scripts.

### 2-UpdateProjectFiles:

* 1-UpdateProjectFiles.ps1 – Find and replace the values in the CSFundamentals Visual Studio solution. Setting starting with “csfundamentals” are replaced using the values from the input files AzureValues.csv and CertThumprints.csv.

### 3-HostedService

* **\*\* Manual step:** If necessary, run 0-Set-AzurePublishSettingsFile.ps1. This updates the Azure default subscription xml data files for Powershell located at “%APPDATA%\Windows Azure Powershell”.
* **\*\* Manual step:** If necessary, upload your Azure “Management Certificate” (.cer file extension) to the Windows Azure Portal. The management certificate is required to execute Powershell scripts against the Azure subscription.
* 1-CreateScheduler\_HostedServices.ps1 – Create the Scheduler Hosted Service and upload the SSL and RDP certificates.
* 2-CreateRouter\_HostedService.ps1 – Create the Router Hosted Service and upload the SSL and RDP certificates.
* 3-CreatePods\_HostedServices.ps1 – Create the CSFundamentalsService Hosted Services and upload the SSL and RDP certificates. The number of pods, or Hosted Services, is determined by the “csfundamentals\_podcount” value in the AzureValues.csv file. For example, “csfundamentals\_podcount,10” will enable 10 CSFundamentalsService Hosted Services to be created.

### 4-RootUserProfileDB

* **\*\* Manual step:** The create database and create objects scripts require sqlcmd.exe to be installed and in your path. Sqlcmd.exe for SQL Server can be downloaded and installed, or it’s installed with SQLEXPRESS or SQL Server 2012 (or previous versions).
* 1-CreateDB.ps1 – Creates the “Root” and “UserProfile” databases in the Azure SQL Database service. The number of “Root” databases created is specified by “csfundamentalsrootdbcount” and the number of “UserProfile” created is specified by “csfundamentalsUserProfledbcount”.
* 2-CreateObjects.ps1 – For each “Root” and “UserProfile” database, creates the tables and stored procedures.

### 5-OpsStatsDB

* 1-CreateDB.ps1 – Creates the “OpsStatsDB” database.
* 2-CreateObjects.ps1 – Creates the tables, functions and views in the “OpsStatsDB”.

### 6-StorageAccount

* 1-CopyXMLFiles.ps1 – Copies 2 runtime configuration files: QuartzJobs.xml used by the SchedulerService and RootUsersShardMap.xml used by both the SchedulerService and CSFundamentalsService.
* **\*\* Manual step:** Verify the shard map in RootUsersShardMap.xml matches the “Root” and “UserProfile” database counts and ranges. The default shard configuration used for “Root” and “UserProfile” databases in RootUsersShardMap.xml are for 2 databases each.
* **\*\* Manual step:** Upload both .xml files to WA Blob Storage Account (create the container, too). To upload the xml files, use a tool of your choice or use Azure Storage Explorer from Neudesic for free found at: <http://azurestorageexplorer.codeplex.com/>. The container name in AzureValues.csv is “csfundamentals\_storagecontainer”.

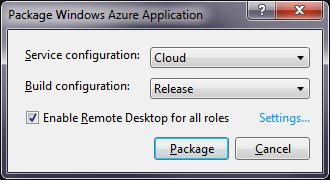
### 7-Build Packages

* **\*\* Manual step:** Build the solution and create the 3 service deployment packages:

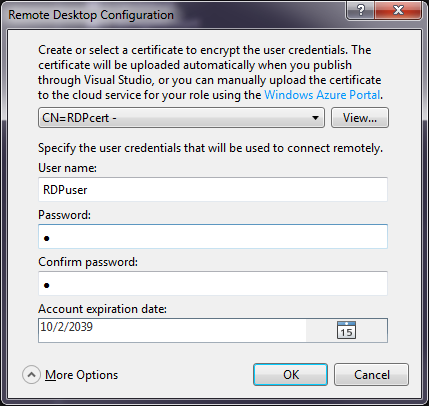
1. CSFundamentals Solution: Select Release BUILD.
2. Enable NuGet packages to be automatically downloaded from the internet (NuGet.org) by righting click on “Solution ‘CSFundamentals’” at the top of the Solution Explorer view. Click on “Enable NuGet Package Restore” menu item and click Yes on the Visual Studio dialog to confirm.
3. RouterService.WebRole: modify project files.

Remove **csfundamentals\_certSSLname** (csfundamentals\_certSSLname.pfx) and update the SSL certificate, in the CSFundamentals.Router.WebRole project. Steps:

1. Delete the csfundamentals\_certSSLname.pfx from the project
2. Add Existing item: find your SSL .PFX file and add it to the project
3. After the .PFX file is added, right click the .PFX file and select properties
4. For “Copy to Output Directory” select “***Copy always***”
5. To build the solution, in Visual Studio click on “BUILD” menu item and select Rebuild Solution.
6. Create the Packages for all 3 services, SchedulerService, RouterService and CSFundamentalsService:
   * For each service, right-click the service project file and select Package to bring up the dialog below:



* Verify the configurations for the Service and Build are Cloud and Release, respectively. Check the “Enable Remote Desktop for all roles” and click the blue “Settings…” link to open the Remote Desktop Configuration dialog.



* For the remote desktop user (RDPuser, for example), enter the password and confirm it then click OK. Now click the Package button to start the package generation. Again, follow these steps to build the Packages for all 3 services, SchedulerService, RouterService and CSFundamentalsService.

### 8-Deploy

* 1-CopyDeploymentFiles.ps1 – Copy the “Release” build packages for each of the 3 services created in Visual Studio CloudServiceFundamentals.sln solution.
* 2-SchedulerServiceDeploy.ps1 – Deploy the SchedulerService to Windows Azure using the files in the SchedulerService directory: SchedulerService.cspkg and ServiceConfiguration.Cloud.cscfg.
* 3-RouterDeploy.ps1 – Deploy the RouterService to Windows Azure using the files in the Router directory: RouterService.cspkg and ServiceConfiguration.Cloud.cscfg.
* 4-PodsDeploy.ps1 – Deploy the CSFundamentalsService to Windows Azure using the files in the Pod directory: CSFundamentalsService.cspkg and ServiceConfiguration.Cloud.cscfg.

# Appendix: Testing and Validation

## Test Environment

Testing of the CSFundamentals solution can be accomplished by using Visual Studio 2012 test harness for both functional and load testing scenarios. The test environment with Visual Studio 2012 components was composed of 1 VSController agent and 40 VSAgents.

## Benchmarking the CSFundamentals Application

The CSFundamentals application provides functionality for three key scenarios:

1. Registration of new users
2. Signing in users (login)
3. Add Comments to a user’s profile

## Benchmark Results

Each of these scenarios were load tested in Windows Azure using the test harness previously described. The application sustained 1 million unique registrations over a 5 minute period with about 10,000 test threads simulating the 1 million users. Approximately the same throughput was measured for the login and comments scenarios.

## Benchmark Azure Configuration

The CSFundamentals application benchmarking configuration:

* RouterService: 40 Medium instances
* CSFundamentals – 2 pods/hosted services with *each*:
  + Web Roles: 40 Medium instances
  + Worker Roles: 4 Large instances for Azure Caching
* SchedulerService: 1 Small instance
* Databases for CSFundamentals:
  + 40 Root databases
  + 40 UserProfile databases

# Appendix B: Azure Subscription Configuration Settings

Use the table below as a checklist for your Azure subscription values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Purpose** | **Description** | **CSFundamentals Solution Tags** | **Your values here** |
| Subscription | Subscription Id | csfundamentals\_subscriptionid |  |
|  | Subscription Name | csfundamentals\_subscriptionname |  |
| Router/Pod | Hosted Services | csfundamentals\_dnsname |  |
|  | Pod Service Count | csfundamentals\_podcount |  |
| Scheduler Service | Hosted Services | csfundamentalsscheduler\_dnsname |  |
| Storage | Storage Account Name | csfundamentals\_storageaccountname |  |
|  | Storage Account Key | csfundamentals\_storageaccountkey |  |
|  | Storage Container | csfundamentals\_storagecontainer |  |
| Certificates | Mgmt Certificate Name | csfundamentals\_certMGMTname |  |
|  | Mgmt Certificate Password | csfundamentals\_certMGMTpwd |  |
|  | SSL Certificate Name | csfundamentals\_certSSLname |  |
|  | SSL Certificate Password | csfundamentals\_certSSLpwd |  |
|  | SSL Certificate Thumbprint | csfundamentals\_certSSLthumbprint |  |
|  | RDP Certificate Name | csfundamentals\_certRDPname |  |
|  | RDP Certificate Password | csfundamentals\_certRDPpwd |  |
|  | RDP Certificate Thumbprint | csfundamentals\_certRDPthumbprint |  |
| rdpuser | RDP User | csfundamentals\_RDPuser |  |
|  | RDP Password | csfundamentals\_RDPpwd |  |
| Database | DB Server | csfundamentals\_dbservername |  |
|  | DB UID | csfundamentals\_dbusername |  |
|  | DB PWD | csfundamentals\_dbpassword |  |
|  | Root DB Count | csfundamentalsrootdbcount |  |
|  | Root DB Name (prefix) | CSFundamentalsRootDB |  |
|  | UserProfile DB Count | csfundamentalsuserprofledbcount |  |
|  | UserProfile DB Name (prefix) | CSFundamentalsUserProfileDB |  |
| Azure Cache | Worker Role Name | CSFundamentalsCache.WorkerRole |  |
| Datacenter | WA DC Name | csfundamentals\_walocation |  |