Sage 300 Web Screens SDK

Software Architecture

August 2021

This is a publication of Sage Software, Inc.

Copyright © 2021. Sage Software, Inc. All rights reserved.

Sage, the Sage logos, and the Sage product and service names mentioned herein are registered trademarks or trademarks of Sage Software, Inc. or its affiliated entities. All other trademarks are the property of their respective owners.

Business Objects® and the Business Objects logo, BusinessObjects®, and Crystal Reports® are trademarks or registered trademarks of Business Objects Software Ltd. in the United States and in other countries. Business Objects is an SAP company.

Microsoft ®SQL Server®, Windows Vista® and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or in other countries. The names of all other products and services are property of their respective owners.

Your use of this document and the Sage product(s) described herein is governed by the terms and conditions of the Sage End User License Agreement ("EULA") or other agreement that is provided with or included in the Sage product. Nothing in this document supplements, modifies or amends those terms and conditions. Except as expressly stated in those terms and conditions, the information in this document is provided by Sage "AS IS" and Sage disclaims all express, implied or statutory warranties of any kind, including but not limited to the warranties of merchantability, fitness for a particular purpose or of noninfringement.

No implied license is granted to you under any Sage intellectual property or trade secret right. Sage reserves the right to revise, supplement or remove information in this document at any time without notice to you or others.

Sage End User License: [www.sage.com/us/legal/eula](http://www.sage.com/us/legal/eula)

Contents

[1. Introduction 5](#_Toc445305356)

[2. Technical Requirements 6](#_Toc445305357)

[3. Technical Solution 7](#_Toc445305358)

[3.1 Development Technology 7](#_Toc445305359)

[3.2 Architecture Views 8](#_Toc445305360)

[3.3 Logical view 8](#_Toc445305361)

[3.3.1 Presentation Layer 9](#_Toc445305362)

[3.3.2 Service Layer (Web API) 11](#_Toc445305363)

[3.3.3 Screen definitions 12](#_Toc445305364)

[3.3.4 Business Layer 12](#_Toc445305365)

[3.3.5 Wrapper Layer 14](#_Toc445305366)

[3.3.6 Interop Layer 15](#_Toc445305367)

[3.4 Deployment View 16](#_Toc445305368)

[3.4.1 Deployment View – Windows Azure 16](#_Toc445305369)

[3.4.2 Deployment View – On Premise 20](#_Toc445305370)

[3.5 SDK 21](#_Toc445305371)

[3.6 Customizations 21](#_Toc445305372)

[3.6.1 OData RESTful Web Services 21](#_Toc445305373)

[3.6.2 Programmatic Customizations 22](#_Toc445305374)

[3.6.3 Metadata Driven Customization-Field Level 22](#_Toc445305375)

[3.6.4 Simple Screen Modifications 24](#_Toc445305376)

[3.7 Cross Cutting Concerns 24](#_Toc445305377)

[3.7.1 Security 24](#_Toc445305378)

[3.7.2 Performance 25](#_Toc445305379)

[3.7.3 Multi-tenancy 25](#_Toc445305380)

[3.7.4 Dependency Injection (DI) 25](#_Toc445305381)

[3.7.5 Unit Tests and Mocking 26](#_Toc445305382)

[3.7.6 Globalization and localization 26](#_Toc445305383)

[3.8 Reports 27](#_Toc445305384)

[3.9 Automation 27](#_Toc445305385)

[3.10 Productivity 28](#_Toc445305386)

[3.11 Guidelines 28](#_Toc445305387)

[3.12 Future Enhancements 29](#_Toc445305388)

[3.12.1 64 Bit 29](#_Toc445305389)

[3.12.2 Unicode 29](#_Toc445305390)

[3.12.3 Sage 300 System Manager Limits 30](#_Toc445305391)

[3.12.4 Remove Shared Data Folder 30](#_Toc445305392)

[3.13 Definitions, Acronyms, and Abbreviations 30](#_Toc445305393)

[3.14 References 30](#_Toc445305394)

1. Introduction

Sage 300 has always been a strong three-tier Microsoft Windows desktop accounting application. A project was undertaken to replace the User Interface (“UI”) layer written in Visual Basic 6 with a more modern web-based UI layer that would run in any browser and consist entirely of HTML, JavaScript, and CSS.

This Software Architecture Document provides a comprehensive architectural overview of the Sage 300 application in regard to these new web-based UIs (also called “web screens” here and in other documentation).

This document also provides a high-level description of the goals of the architecture, the features provided by the system, and the architectural styles and components that have been selected to implement the features.

A number of different architectural views are presented to depict different aspects of the system, and to capture and convey the significant architectural decisions that have been made in designing the system.

1. Technical Requirements

This section describes technical requirements for Sage 300 web screens.

Objectives of the Web UI project include:

* Modernize Sage 300 UI to replace VB6 screens with modern appealing supported technology that provides a great customer experience.
* Evolve toward an extensible architecture
* Be able to fully run under PaaS as a scalable fault-tolerant automatically managed system, and avoid use of IaaS elements.
* Expose Sage 300 functionality via modern industry-standard web service interface.
* Make the application deployable on-premise by using different configuration files.
* Make different deployments available through customization.

Technical requirements were collected in the following areas:

* Development Technology
* Customizations
* Deployment
* RESTful services
* Performance
* Multi-tenancy

Requirements for the above areas are detailed in this document.

Assumption

* Business Views, Database Driver, and Database Schema will be used “as is”.

1. Technical Solution

The technical solution for implementation of Sage 300 includes development of a multi-tenant ASP.Net MVC 5 application in the .Net 4.5 framework.

RESTful web services are also exposed for extensibility.

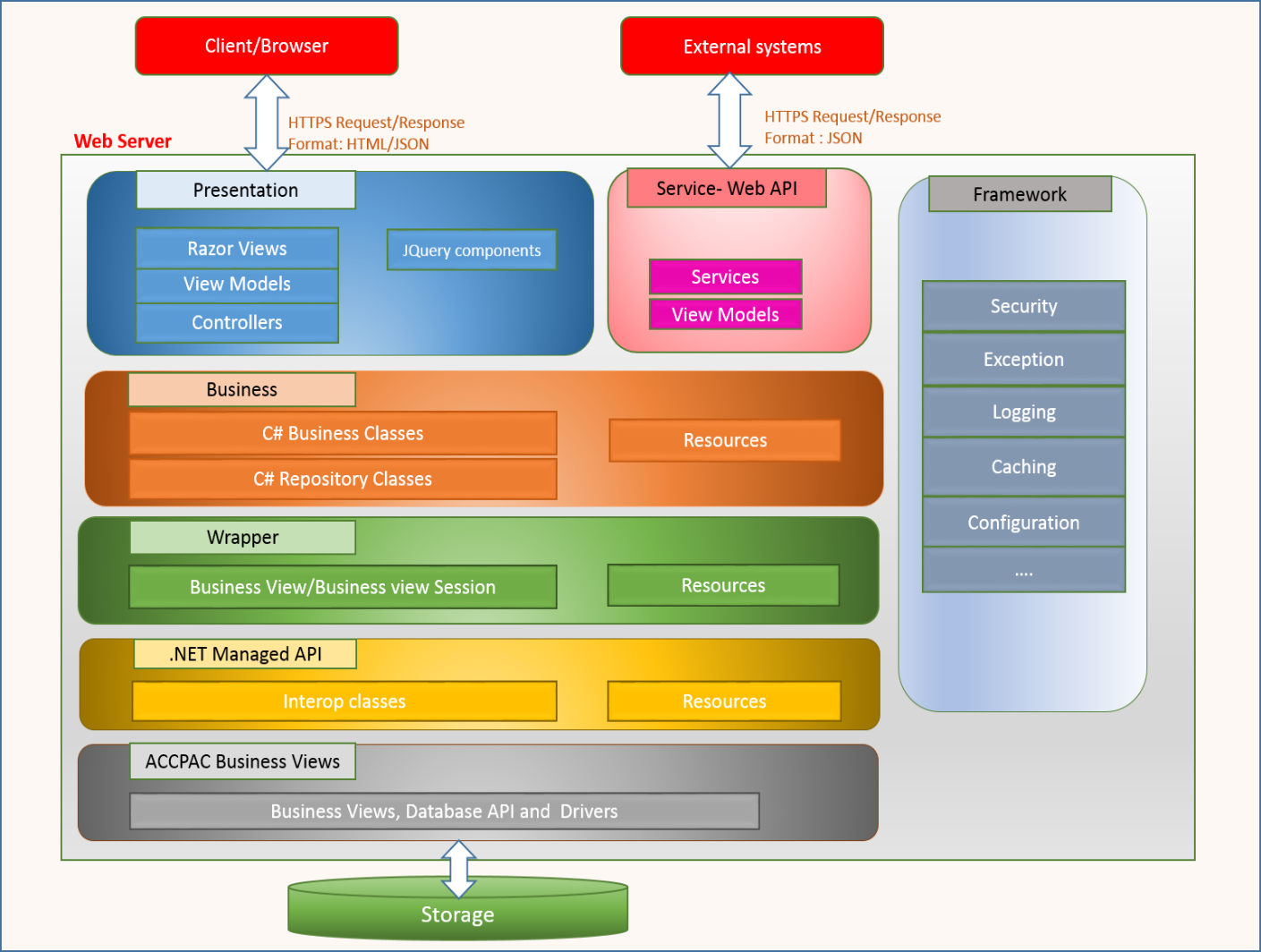
* 1. Development Technology

The following technology has been adopted for the development of Sage 300 web screens.

* ASP.Net MVC5 framework
* C# development language
* HTML5, JavaScript (JQuery), Knockout, CSS
* .Net 4.8.2 framework
* Visual Studio 2019
* IIS 8.5
* Application Request Routing 3.0
* Kendo UI widgets
* Unity Framework–Dependency Injection
* Enterprise Library Logging Application Block
* Entity Framework 5 ORM
* Automation-Selenium Web Driver
* Windows Azure Cloud Platform
* Windows Azure SDK 2.3
* Windows Azure SQL Database
* Table, Blob Storage Services, Queues
* Web Role
* Worker Role
* Role Cache
* Transient Fault Handling Application Block
* Auto Scaling Application Block
* On premise
* SQL Server 2012
* Windows Server 2012
  1. Architecture Views

This section describes the technical solution in detail using the Logical and Deployment views of the architecture. The cross cutting concerns of the application are also covered.

* 1. Logical view



The application is divided into layers for ensuring separation of concerns (SOC). The above diagram provides direction to application designers to keep logically related components in separate layers, and describes how they communicate with each other.

The adoption of logically related components in architecture layers helps promote complex operational requirements with good maintainability, reusability, scalability, and security.

The layering in the architecture provides the following advantages:

1. Changes in one part of the solution will impact other parts minimally, reducing the work required to repair defects, enhancing application maintenance, and improving the overall flexibility of the application.
2. Separation of responsibilities/concerns between components increases flexibility, maintainability, and scalability.
3. Many components can be reused in different modules of the application.
4. Development teams should be able to work on different parts of the solution .
5. Components that are not directly related can be loosely coupled.
6. The different components in a solution must be able to be deployed independently, and maintained and updated at different times.
   * 1. Presentation Layer

3.3.1.1 Server

The presentation layer is developed using ASP.Net MVC 5 Controllers and Razor Views. The presentation layer follows the Model-View-Controller pattern to implement the user interface.

The Model-View-Controller (MVC) architectural pattern separates an application into three main components: the model, the view, and the controller. The ASP.NET MVC framework is a lightweight and highly testable presentation framework.

The MVC framework includes the following components:

[](http://i1.asp.net/asp.net/images/mvc/21/image001.png)

* Models (or View Models)

Model objects are the parts of the application that implement the logic for the application’s data. These will be serializable POCO classes.

* Views

Views are the components that display the application’s user interface (UI). Views will be developed using Razor syntax, HTML5, CSS, JQuery, and Kendo UI.

* Controllers

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render that displays the UI.

The application is divided into areas. Each area has self-contained routes, controllers, and views.

All UI validation (required fields, maximum length, regular expressions, and so on) is implemented using Data Annotations in the view models. This enables the application to perform both client- and server-side validation.

Custom HtmlHelpers or HtmlHelper extensions are used in the razor views to generate customized HTML output.

Partial Views have been created for reusable controls such as Address.

For additional information about customization, see **3.6 - Customizations**.

3.3.1.2 Client

To provide responsive, and interactive experience, the application manages client-side structure changes without performing full-page reloads.

The initial HTML markup is generated by the web server. After generating the initial server HTML, the client side JavaScript responds to user gestures, requests data (or granular HTML fragments), and initiates UI changes without posting back to the server.

* Kendo UI widgets are used wherever applicable.
* jQuery, a cross-browser JavaScript library, is used for client-side scripting.
* Knockout.js is used for data binding.

**Modularity**

There is clear separation between HTML markup and JavaScript files.

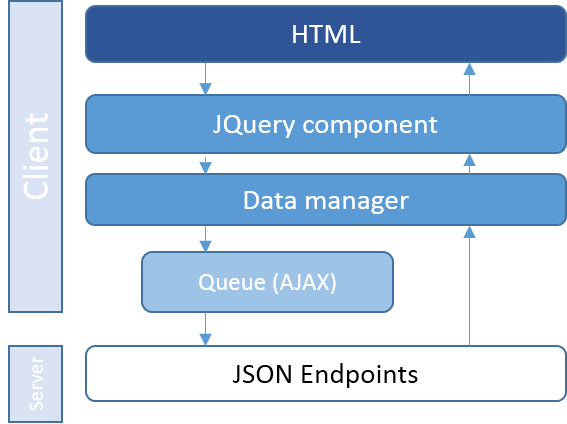
There is no script inside the Razor (.cshtml) files. Each view has a different JavaScript file. This will increase testability, ease maintenance, and enable reuse.

**Communication and Data**

All data requests are made via Ajax and are routed through the data manager. Having a single object handle data requests simplifies the client-side calling code, improves application testability, and facilitates cleaner application evolution when client-side libraries advance or change.

The single data manager object also affords you the opportunity to implement client-side data caching in a central location. Data can be cached in a JavaScript object, or using HTML5 local storage if required.

Whenever there is an Ajax call, the user is not blocked from entering data unless absolutely required. Ajax requests are sequential in order to ensure that functionality behaves correctly.



The following plugins are used to achieve this:

* <https://github.com/Sage/streamlinejs>
* <https://github.com/Foliotek/ajaxq>

Where UIs include tabbed navigation, initial data is loaded only for the current tab and those tabs for which there is a small, manageable amount of data. If there are tabs for which data is substantial (for example, a table with many detail records), data is loaded only when the user clicks on that tab.

**Browser support**

The presentation layer uses HTML5, which is available for all current browsers. The application supports the latest stable version of all major browsers, including Internet Explorer, Google Chrome, Apple Safari, and Mozilla Firefox.

* + 1. Service Layer (Web API)

This layer contains RESTful interfaces and components to handle and process OData requests.

This layer interprets RESTful HTTP requests and passes requests to the Business layer to implement the required functionality.

The architecture leverages the OData support built into the Microsoft technology stack. A LINQ provider was written to translate LINQ requests to business layer requests to support standard OData queries.

The view models sublayer in this layer represents the data that are returned and received from the client applications. View models are expected to contain only the data and relationship with other types in this layer. The view models are serializable POCO types.

This layer communicates with the business layer. Third-party applications (ISV) can consume these services to extend the functionality.

* + 1. Screen definitions
* Razor syntax is a declarative language that can provide the screen definition. However, the end user may not understand this, as it can include customized Razor syntax or partial views.
* HTML is also a declarative language that can provide the screen definition. This can be generated from the Razor view and the model. But this can also include CSS, custom HTML attributes, and scripts tags that will be difficult for the end user to understand.
  + 1. Business Layer

This layer contains all the business components that implement the business logic to retrieve required data from the Sage 300 Business Views. Each component is responsible for a particular functionality.

Repository classes are responsible for managing the business views (for example, opening the business views, retrieving or saving the data, and disposing the business views). These classes aggregate multiple business views into a logical entity.

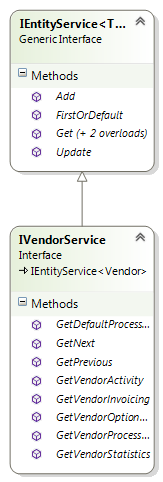
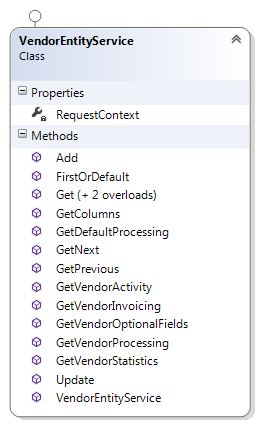
These business components return Model objects to the web application or Service layer.

All business validations are implemented in Sage 300 business views. All validation exceptions are captured in the Sage 300 Session object, and are displayed to the user. Additional business logic that is not currently in the Business Views (C DLL files) is implemented here if required.

Transactions are handled in the Sage 300 business views. This layer does not start, rollback, or commit any transactions.

These components are provided as a separate SDK that can be used by third-party vendors to extend functionality.

Below is the class diagram for a vendor service.

* IEntityService: This interface is implemented across all services. This ensures the APIs are consistent, and makes the service layer easy to use.
* IVendorService: This interface has specific methods relevant to vendors, and implements IEntityService
* VendorEntityService: This implements IVendorService, and has concrete implementation.

**Stateful flow**

In most cases, the application is stateless. In some scenarios (such as transaction screens), there are times when application needs to be stateful. Where a stateful session is required, caching of the Sage 300 business views is done in this layer in repository classes.

Because Sage 300 business views cannot be serialized, there is stickiness between the client and the server. Stickiness between client and server is achieved using IIS ARR. Sage 300 business views are cached in the same instance of the web role, using the System.Runtime.Caching namespace.

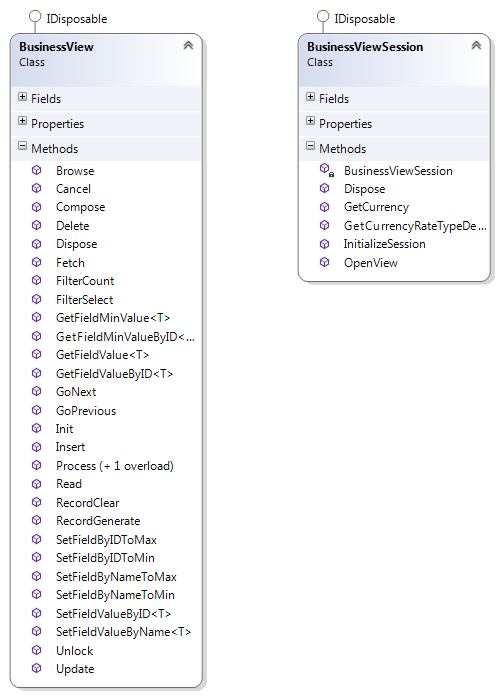
A typical flow (A/P invoice) is as follows:

* Whenever there is a full page load (not an Ajax call), a Guid will be generated on the server and will be sent to the client. This Guid will be used as the cache ID.
* The user enters the batch number and the first invoice is loaded by making an Ajax request. The request uses the Guid created above. The server caches the session and business objects and maps the business view cache object with the above generated key. This Sage 300 business view is cached on the web role.
* Whenever server calls are made, whatever changes are done in UI (delta) will be sent to the server and server will retrieve the business views from the cache based on the Guid and will update the business view based on the changes. Server will send the response and UI will be updated.
* Once the save/cancel is called again business view will be retrieved from the cache and will be updated. Once the transaction is completed/roll backed business view object will be disposed. At this time the cache will also be removed.

For stateless, Sage 300 business view will be disposed once the data is retrieved or saved.

* + 1. Wrapper Layer

This layer provides the wrapper classes, which provide an abstraction to access the Interop layer. Having this layer removes the dependency on using any particular implementation of the Interop layer.



* + 1. Interop Layer

Sage 300 Business Logic is implemented through a number of standard APIs that are part of the Sage 300 System Manager.

* Individual Business Logic objects are called Views.
* Each View is implemented as a Windows DLL and is written in C.
* The application uses a .Net library (ACCPAC.NET) which calls these DLLs through a COM interface that communicates with RotoView.
  1. Deployment View

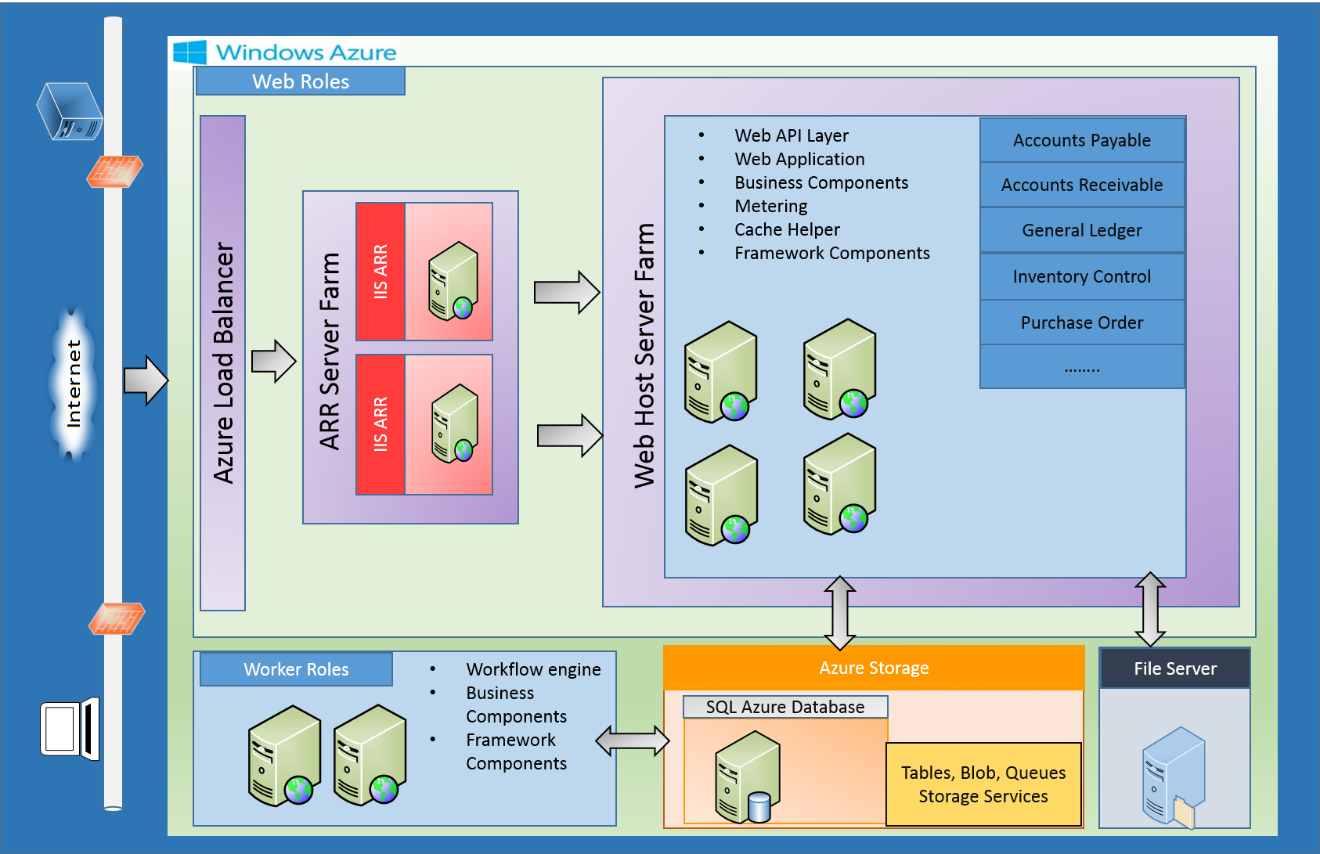
The application has a single code base, which can be deployed on the cloud as well as on-premises. The application implements the provider model, which allows different implementation of components based on the deployment.

Cloud and on-premises deployment have different component configurations.

* + 1. Deployment View – Windows Azure

The following diagram illustrates the physical deployment view of the application on the Windows Azure platform.

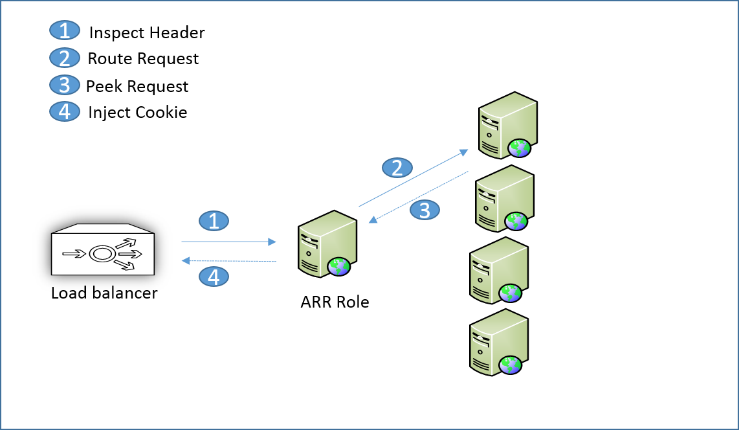
Note: The one non-PaaS component, the file server, has been replaced by the Azure File service.



Infrastructure is as follows:

Web Role

* The web role has the MVC web application hosted on IIS 8.5.
* The web role contains an endpoint for externally accessed RESTful services. A separate web role provides the RESTful services.
* Separate web roles have IIS Application Request Routing (ARR), which enables sticky sessions using client affinity. The first request injects the cookie; the rest of the calls are diverted to the same server via the cookie. The cookie contains information regarding the server to which the request is to be forwarded.



File Server – Now Azure Files

The file server has been mostly replaced by the Azure Files PaaS service. This section remains for the purpose of documenting the small number of files that are still maintained on the file server.

For each tenant, the file server contains a share, to which a web role setup's Shared Data is mapped. All ISM files, license files, and so on are stored in this location.

|  |  |  |
| --- | --- | --- |
| File Name | Required changes | Comments |
| Lanpak.bin | No changes required. | Used for multi-user access. Uses virtual locks on the file. This is used by the Session object. |
| Semaphore.bin | No changes required. | Used for locks that are applied. |
| Orgs.ism | No changes required. | Used for storing the list of database connections. |
| Cust.ism | Changes may be required based on the approach. Custom reports may be uploaded to the blob storage. | Used for customizing reports.  Contains the path for custom report storage. |
| MakFils.ism | No changes required. | Used for signing in. |
| Restart.ism | No changes required. | Used for retrying a failed action (for selected actions such as import/export) from the position it failed.  Note: This is deprecated. No new functionality will use this file, but it is still used in older code. |
| Colorconfigure.ism | No changes required. | Used to differentiate the title bar of the Sage 300 application based on company. |
| Browse.ism | No changes required. | Used to store extra information about the users stored in users.ism. |
| Users.ism | Changes may be required to have a mapping to Sage ID. A table must be created in SQL Azure to map Sage ID to the User ID. | Used to store lists of users.  When Open Session is invoked, the data is retrieved from this file. |
| Winuser.ism | No changes required. | Used to store a list of users when the system allows signing in with a Windows AD account. This is required only for on-premises deployment. |
| <Company>\_C.ism | No changes required. | Used to store customizations for the company. |
| <Company>\_P.ism | New table needs to be created. | Used to store all preferences of users for the company, such as table column width, columns to be displayed, and so on. |
| License files | No changes required. | When the UI runs, API calls are available to check if the signed in user has access to functionality. This information is stored in LIC files. |

Azure Table Storage

Because Azure table storage is faster and less expensive than SQL Azure storage, we plan to store large quantities of diagnostic data here, as well as audit tables and any other data that does not require relational storage or queries.

Worker Role and Queues

Azure Queues are used with worker roles for asynchronous processing. Any processing screen, such as the Create New Year screen, is processed using worker roles.

SQL Database

SQL Azure is used for all relational data needs. Each tenant has its own databases.

Caching

* Sage 300 Business views

All Sage 300 business views are cached using the System.Runtime.Caching namespace in the web role.

Note: Because Sage 300 business views are not serializable CLR objects, they cannot use Azure Role based caching.

* Common Language Runtime (CLR) objects

ASP.NET applications use this caching for the common scenarios of session state, lookup data, and output caching.

There are two main deployment topologies for caching: dedicated and co-located. In the dedicated topology, you define a worker role that is dedicated to caching. This means that all of the worker role's available memory is used for the caching and operating overhead. In general, a dedicated caching role provides the best performance, because it does not share the role's virtual machine with any other application services. It also provides the most flexibility, because you can scale the caching role independently. For these reasons, using a dedicated topology is the recommended caching architecture.

In a co-located topology, you use a percentage of available memory on application roles for caching. For example, you could assign 20% of the physical memory for caching on each web role instance.

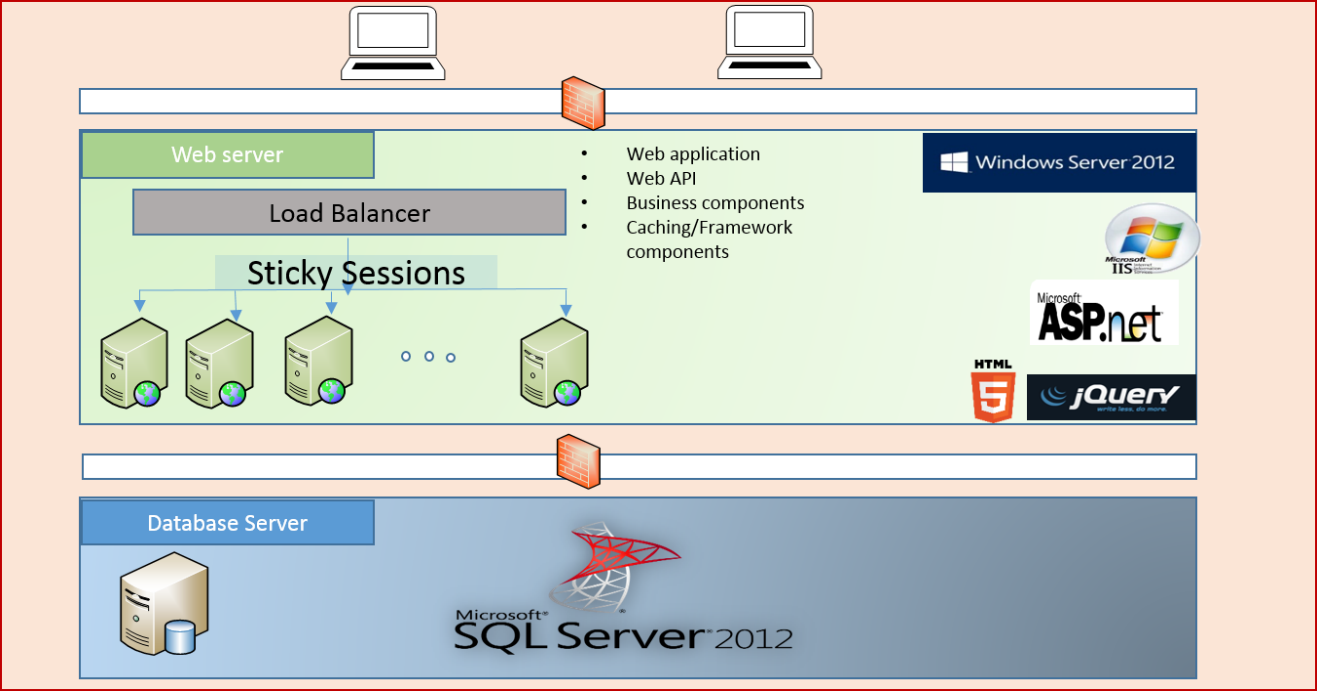
Logging

Windows Azure Diagnostics enables you to collect diagnostic data from an application running in Windows Azure. We use Azure Diagnostics with the Enterprise application logging block to implement logging and tracing.

These logs will be stored in Windows Azure table storage (Wadlog tables).

* + 1. Deployment View – On Premise

The following diagram illustrates the physical deployment view of the on-premises application on the Windows platform. The application can be deployed on a single web server or have multiple web servers and use load balancing with sticky sessions, allowing the application to scale horizontally.



* Application Server: The application is hosted on Windows Server 2012 and IIS 8.0.
* SQL Database: SQL Server 2012 is used for all relational data needs.
* Caching: In-memory caching is implemented using the System.Runtime.Caching namespace.

We can also have dedicated caching for lookup data, output caching, and so on. AppFabric provides dedicated caching capabilities to provide high-speed access, scale, and high availability to application data.

* Logging: The Enterprise Library Logging Application Block is used for logging.
* Authentication: This should work as-is, but will not support Sage ID.
  1. SDK

Sage allows third-party development partners to add their own accounting modules to the Sage 300 system. These products behave exactly like the modules developed by Sage. We have always provided an SDK to allow third-party parties to achieve this.

For the Web UIs, we also provide a SDK in the form of a Visual Studio extension that allows you to create a Web UI solution (for an Accounting module) and then create individual UIs within that solution.

This Visual Studio extension complements the existing SDK, which provides the mechanisms to create your Sage 300 Business Logic, Crystal Reports, Database Schema, and other necessary Sage 300 components.

The Web UI framework is architected so third-party solutions can be added simply by copying files to a predefined location. It does not require any modification of existing files that have been installed by Sage 300.

We discover each application by searching the Sage300\Online\Web folder for files matching the pattern xxbootstrapper.xml, where xx is the developer’s unique two-character prefix. The xxbootstrapper.xml file defines the DLLs that need to be dynamically loaded via dependency injection to become part of the system.

* 1. Customizations

Sage 300 has a very powerful customization model accomplished via VBA macros. Similar functionality is required for the Web UIs. However, we must be careful to support continuous upgrades, and cannot introduce any customization technology that would provide friction to upgrades.

Similarly, all customizations should be appropriate to both cloud and on-premises deployment. However, practically speaking, we will need to allow a bit more freedom in the on-premises world to provide custom DLLs and programming that the cloud would not allow.

|  |  |  |
| --- | --- | --- |
| Customizations | On Premise | On Cloud |
| Plugin model for third-party components | Allowed | n.a. |
| Metadata customization (codeless) | Allowed | Allowed |
| Screen layout customization | Allowed | Allowed |

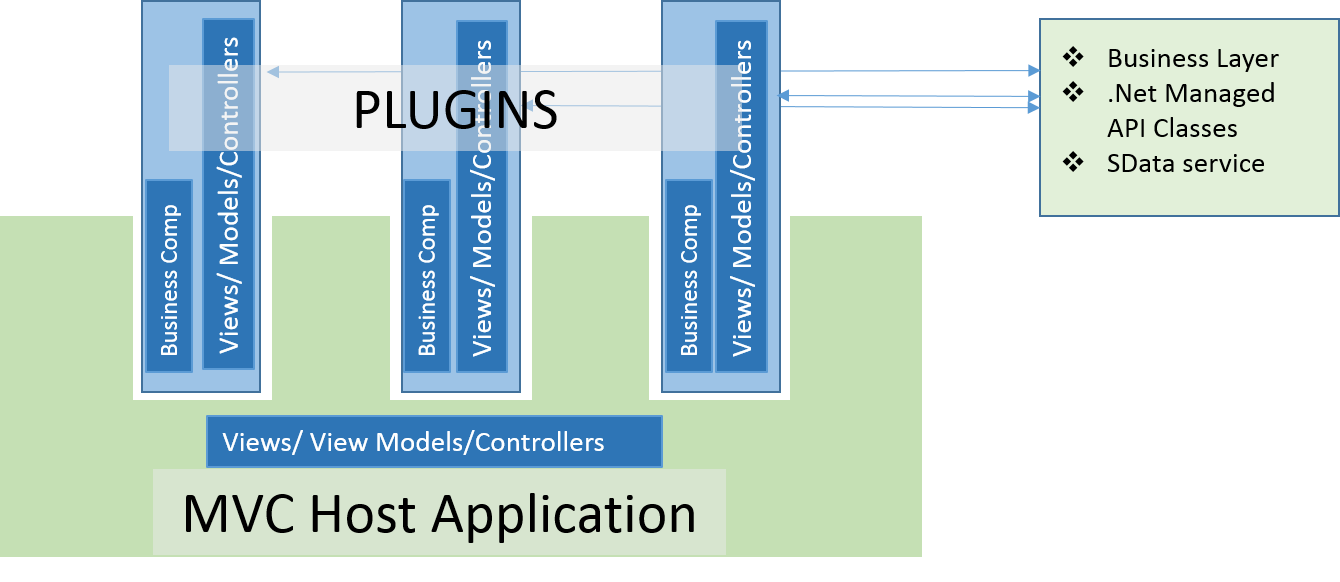
* + 1. OData RESTful Web Services

Partners can use ODATA services to interact with the business logic. This is the primary integration point for getting data into and out of the system, and is ideal for integrations to cloud-based systems.

* + 1. Programmatic Customizations

Using Unity interception, we can allow dynamically loaded DLLs provided by third parties to be called when various event happen within the system. This is basically a plugin architecture.

If a developer needs to provide extra validations or custom default values, they can provide such a plugin, and then intercept and modify the data for the correct events. This is similar to intercepting datasource events in VB to do custom processing.



Plugins can be developed in a standard MVC project, and may use all the existing infrastructure of the ASP.NET MVC framework.

Plugin assemblies are loaded into the main MVC Application domain. Plugins have the same trust level as the host.

* + 1. Metadata Driven Customization-Field Level

The following are examples of uses of metadata-driven customization at the field level:

* Make a field required
* Hide unused fields
* Change some fields to read-only.
* Change labels to match company terminology
* Change the tab order of the controls to skip infrequently used fields
* Add extra validation
* Provide custom defaults for fields
* Add extra warnings and validation errors

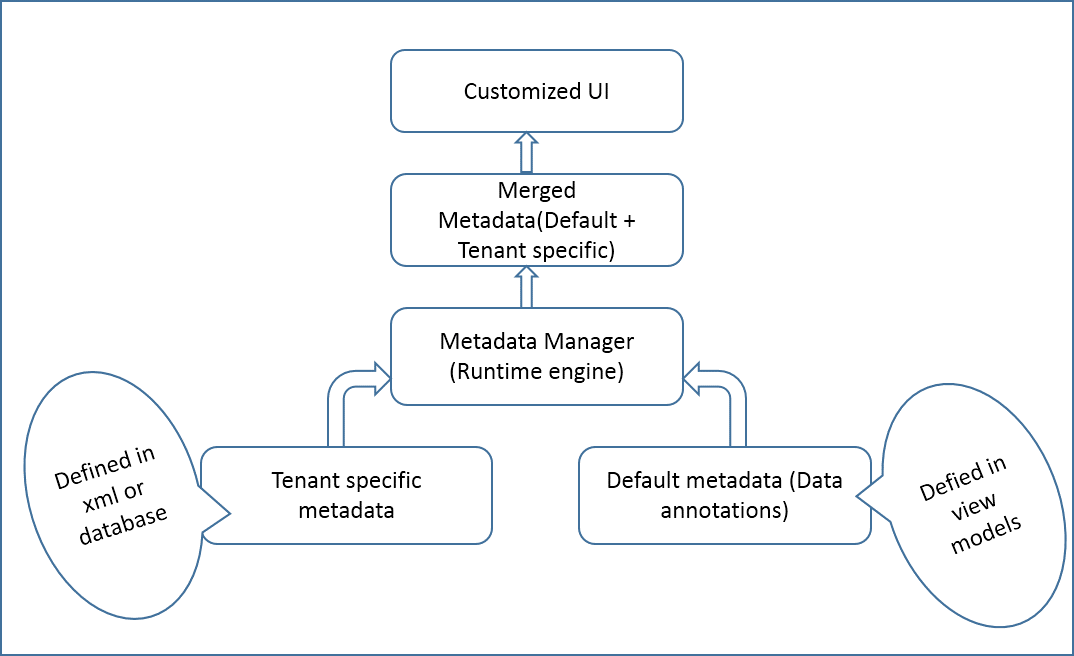
These can be implemented by storing metadata specific to the tenant in the database. Metadata can be defined in the XML format, which can be stored in the database.

The application can be developed to generate the XML document (metadata).

Data annotations will provide default validations based on the business views.

The Runtime Metadata Manager will combine the tenant specific metadata defined in the database and default metadata and will render the customized UI.

This approach will provide both server and client side validation.



For example, the Vendor model can have the following validations by default:

|  |
| --- |
| /// <summary>  /// Vendor Number  /// </summary>  [Required]  [StringLength(11)]  public string VendorNo { get; set; }  /// <summary>  /// Vendor Name  /// </summary>  public string VendorName { get; set; } |

If the application needs to be customized to make the Vendor Name field required, tenant-specific metadata can be injected at runtime by extending DataAnnotationsModelValidatorProvider.

If tenant-specific metadata is in an XML file:

|  |
| --- |
| <Model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="Vendor">  <property name="VendorName">  <Validations>  <validation type="Required">validation>  </Validations>  </property>  </Model> |

The metadata manager will read the XML file at runtime and inject the data annotations (RequiredField annotation) in the model before rendering the view.

A list of attributes is provided in MVC:

<http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx>

Custom attributes can also be created to implement any specific behavior or validation.

* + 1. Simple Screen Modifications

General customization of the Razor Views will break upgrades and is not allowed.

More indirect screen customizations are acceptable. For example, if you need to add a control, rather than adding it directly to the Razor view, you specify that you want a control of type (such as a button) to the left of another control (such as the Save button).

This way, these changes can be applied after an upgrade without having to be manually applied to a new Razor view.

* 1. Cross Cutting Concerns
     1. Security

Authentication

* Cloud deployment: Sage ID integration is required.
* On-premises deployment: The current authentication model to be supported.

Authorization

Authorization is handled by the Sage 300 Business Views and will work as-is. The UI will not provide links or access to those functionalities for which users are not authorized.

Database Connection

There can be multiple databases for a tenant.

Each company has a different database.

There will be only one System database per tenant.

Database connection information is in orgs.ism, located under in the SITE folder. The system reads the registry entry for ..\ACCPAC\Configuration\ SharedData to find the SITE folder.

Communication

The application is enabled for SSL for all pages to secure the transformation of the information between the client and the server for browser access.

* + 1. Performance

Screens should perform at a reasonable speed that matches customer expectations for performance of a browser application. Data should load and field validation should be performed with a minimum of noticeable lag time.

Pay particular attention to the following performance concerns:

* The application should minimize the number of requests and keep the payload size small.
* Pagination should be enabled for larger data sets.
* Data should be cached on the client and server side wherever applicable.
* Client-side optimizations, such as Minify JavaScript and bundling, should be implemented
  + 1. Multi-tenancy

This application architecture addresses multi-tenancy as follows:

* Use URL routing for tenant identification.
* Use URL routing for company identification.

If information required is across the companies (for example, if the user needs to be presented with a list of companies based on tenant), company will not be part of the URL.

* Use a different database for each company in a tenant.
  + 1. Dependency Injection (DI)

Dependency Injection (DI), or Inversion of Control, is a software design pattern that implements inversion of control for resolving dependencies. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it.

This design pattern promotes loose coupling and improves extensibility of the application.

Adoption of DI is greatly simplified by using DI Container tools, as most of the work (Object creation or Dependent Object Injection, and so on) are boilerplate code and available as part of the Enterprise Library (Unity framework).

The use of DI is recommended in the following two areas:

1. Web Application or Web API Controller Object instantiation.
2. Business Service instantiation

The following requirements apply for using DI on any class:

1. The instance of the class that needs dependent objects to be injected should be instantiated using Unity Container, and all dependent objects should be parameters to the constructor.
2. The class being injected should implement an Interface and should be registered with Unity container.

With the above two requirements met, the approaches described below would be adopted.

Web Application or Web API Controller class instantiation

ASP.NET MVC5 Controller classes are instantiated using ControllerFactory class, and are extensible to register a Dependency Injection tool.

The following steps must be performed:

1. Dependency Injection tool container is registered to ASP.NET MVC4 using DependencyResolver.SetResolver method call passing Unity container instance in Application\_start event.
2. All classes that need to be injected registered to Unity Container in Application start event.
3. All API Container class constructors take parameters (Interfaces of Dependent types). Examples are Business Service types, Application Execution Context, and so on.
4. ASP.NET MVC4 Infrastructure creates Controller instance for executing HTTP requests using Unity Container and Unity container injects the necessary dependent objects to the Constructor if the interface is registered in the Unity container in Step 2.

**Note:** This mechanism is used for Mocking and/or passing fake objects during Unit Testing. The Unity container registration only needs to be different in Unity container.

* + 1. Unit Tests and Mocking

Unit tests will be created for all layers.

The developed unit tests should provide 85% code coverage.

Unit tests will be created using Microsoft’s unit testing framework. Wherever necessary, MOQ mocking library will be used to create mock objects for unit tests.

* + 1. Globalization and localization

The application should adhere to all best practices to implement globalization and localization.

<http://msdn.microsoft.com/en-us/library/w7x1y988.aspx>

If the culture is not supported, the application will manage the resource fallback mechanism.

Localization

Each user can define their language. The application selects language resource files based on the user’s selection.

* All the UI interface content such as menus, field/control labels, messages, and so on should use content from language resource files.
* The UI should be designed to allow for text expansion when text is translated, and should handle text overflow/wrapping elegantly.

Globalization

* The application should be in Unicode.
* Formatting tasks such as date and time or numeric formatting should be culture-specific. Culture information should be taken from the browser.

Languages currently supported: English, French, Chinese, and Spanish.

* 1. Reports

Reports in Sage 300 are generated using SAP Crystal Reports ®.

The application uses the Crystal Reports Viewer .Net control to render reports. The Razor view cannot directly display Crystal reports using Crystal Reports Viewer; instead. Crystal Report Viewer needs to be used by web form. This can be achieved in the MVC application, as web forms can be included in the MVC application.

The application also uses Crystal Reports DLLs to create PDF documents on the server and export them to the client as an HTTP response.

Considering performance and scalability, different web roles for Reports can be provided.

* 1. Automation

The Selenium Web Driver tool is used for automation testing.

Using this tool supports delivering a well-designed object-oriented API that provides improved support for modern web application testing requirements.

Automation guidelines

* Focus on test coverage of core user scenarios, features, and workflows (as opposed to a focus on “code coverage”).
* Separate test code from UI definition code so that tests are not brittle (for example, Selenium Web Driver Page Object Pattern, Telerik Testing Framework Object Manager, Silk Test Object Map, and so on.)
* Design automation with emphasis on maintainability and readability of test code over abstraction (that is, abstract as much as possible without sacrificing readability and maintainability).
* Drive actual browsers for smoke and regressions tests (as opposed to headless testing).
* Be cross-browser capable and consider all supported browsers. Eventual support for iOS and Android browsers may become a requirement in the future.
* Be capable of parallelization in order to speed up test runs.
* Provide reader-friendly reporting of results.

Technical considerations

* The general preference is for automation code to be in C#, due to widespread familiarity. However, this is a preference, not an absolute requirement.
* Test source code is to be stored in a Git repository.
* Assets for data-driven tests are to be “versionable” by Git (that is, in a text format such as CSV, XML, and so on, rather than binary files such as XLS).
* The Sage 300 UI will use Kendo controls, so ease of use in automating these controls will be considered.
  1. Productivity
* MVC Scaffolding will be used for a productivity enhancer. This can generate the basic startup code. We will develop our customized scaffolders to get the initial code based on the type of UI screen.
* Use T4 templates to generate code wherever required. These can be used to create POCO classes or mapping classes and so on.
* UI templates and reusable components will be created for reuse across the project.
  1. Guidelines

Design Guidelines

1. Keep the design simple
2. Ensure to follow OOP design principles such as Encapsulation, Generalization, and Specialization.
3. Ensure that known design patterns have been explored for the solution, and that antipatterns are not adopted.
4. Adopt the Single Responsibility Principle (SRP): each type should have one and only one responsibility.
5. Adopt SOLID Principles (<http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)>).
6. Adhere to architecture suggested classes.
7. Creating instance of any type for calling a public method should involve Dependency Injection for type instantiating.
8. Design for returning the least possible data to caller.
9. Use correct access modifier (public, private, and so on). Effort should be to use minimum required access modifier.

Coding Standards

The Microsoft All-In-One Code Framework will be used as the coding guideline for native C++ and .NET (C# and VB.NET) programming.

<http://blogs.msdn.com/b/onecode/p/introduction.aspx>

JSLint, a static code analysis tool, will be used in software development for ensuring that JavaScript source code complies with coding rules.

Naming Conventions

The base namespace is as follows:

{Company}.{Country}.{Group}.{ProductCategory}.{ProductName}

|  |  |
| --- | --- |
| Components | Namespace |
| Web API SDATA | Sage.CA.SBS.ERP.SAGE300.Web.Api |
| Web application | Sage.CA.SBS.ERP.SAGE300.Web |
| Business Services | Sage.CA.SBS.ERP.SAGE300.Services |
| Repository | Sage.CA.SBS.ERP.SAGE300.Repository |
| Integration | Sage.CA.SBS.ERP.SAGE300.Integration |
| Core | Sage.CA.SBS.ERP.SAGE300.Core |

* 1. Future Enhancements
     1. 64 Bit

The Sage 300 Business Views are compiled in 32 bit, and using these components in IIS limits the memory usage to 2 GB.

To improve performance and allow access to more memory, we plan to build 64 bit versions of the Sage 300 Business Views after the Sage 300 Business Views are upgraded to be built with VS2013 or VS2015.

* + 1. Unicode

The new Web UIs handle all their data using Unicode. However, the Sage 300 Business Logic is not compiled for Unicode, so all data is translated into double byte characters using the code page of the server.

To be a truly international application, the Sage 300 Business Logic needs to be compiled for Unicode. Note that because all the code is in C/C++, this will be a substantial undertaking.

* + 1. Sage 300 System Manager Limits

The core Sage 300 System Manager components have a number of fixed-size tables that place limitations on the scalability of the system. For example, the number of users that can be served by a single web server is currently limited to about 175 due to a limitation on the semaphore API. Extending these limits will be necessary at some point.

In the case of the semaphore API, it may be rewritten entirely to move it to a SQL mechanism rather than file locks on a file in the shared data folder (which is a scaling problem in itself).

Note: The LanPak system has a similar limitation, but because LanPaks are not used in the Web UIs, this problem does not affect web screens.

* + 1. Remove Shared Data Folder

It has been recommended that we eliminate the shared data folder entirely.

Now that we the landlord database is stored both on-premises and in the cloud, theoretically everything stored here (at least when only running Web UIs) could be moved into either the landlord or company databases. A SQL mechanism will need to be developed to determine how elements are controlled with file and byte locks.

* 1. Definitions, Acronyms, and Abbreviations

Coming soon…

* 1. References

|  |  |  |
| --- | --- | --- |
| # | URL / Document | Description |
| 2 | <http://www.windowsazure.com/en-us/home/features/caching> | Windows Azure Caching |
| 3 | <https://github.com/Sage/streamlinejs> | JavaScript Queuing library |
| 4 | <https://github.com/Foliotek/ajaxq> | JavaScript Queuing library |
| 5 | <http://msdn.microsoft.com/en-us/library/dd460648.aspx> | MEF |