|  |
| --- |
| Sage 300 Web Screens |

Subclassing

Implementation of subclassing architecture for all Sage 300 web screens and Web API

Table of Contents

[Introduction 3](#_Toc156993837)

[Overview 3](#_Toc156993838)

[Definition 3](#_Toc156993839)

[Customization 3](#_Toc156993840)

[Solution 4](#_Toc156993841)

[Strategy 4](#_Toc156993842)

[ModelMapper.cs 5](#_Toc156993843)

[Map Getter 6](#_Toc156993844)

[Map Setter 10](#_Toc156993845)

[MapKey 12](#_Toc156993846)

[BusinessEntityField and GridField 12](#_Toc156993847)

[Customization 13](#_Toc156993848)

[Overview 13](#_Toc156993849)

[Sample 13](#_Toc156993850)

[Web API 15](#_Toc156993851)

[Overview 15](#_Toc156993852)

[Utility and Wizards 16](#_Toc156993853)

[Overview 16](#_Toc156993854)

[Web Subclassing Prep Utility 16](#_Toc156993855)

[Web Subclassing Prep Utility 17](#_Toc156993856)

[Web Subclassing Compilation and Deployment Wizard 20](#_Toc156993857)

Introduction

Overview

The purpose of this document is to describe a solution for missing functionality in the web screens. The subclassing of business views to expose additional fields and functionalies are a core aspect that illustrates the flexibility of the Sage 300 architecture. However, this flexibility is only partially available in the web screens.

Definition

Subclassing can be thought of as Override subclassing and Join subclassing. For this discussion, business logic implemented in the business view will not be discussed as this logic is available to both the Sage 300 desktop and web screens.

The missing functionality is the ability to display and interact with fields added to a business view. When a partner subclasses a business view, these added or joined fields will likely be required to be displayed on a screen for user interaction.

Note, this is not always the case since business views may only be subclassed to gather, generate, and/or discover additional data items that are not required to be seen in Sage screens (i.e., Pacific Tech’s Logger, Peresoft’s Metrics, etc.).

At the core of the current inability is the fact that MVC models in the web screens are mapped to the business view at design or compilation time and are therefore static since they are unaware of the business view being subclassed for additional fields.

Customization

Customization in the desktop can involve both the subclassing of the business view and the inheritence and modification of the screen to display the added fields.

Customizaton also is implemented in the web screens, however, since the added fields are not in the MVC model, the web screens can not apply a customization package to display the added fields. Customization will play the central role in displaying the fields once they are added to the MVC model.

Let’s look at the solution.

Solution

Strategy

Since the MVC model was generated at compilation time and not instantiation time, we will need a solution that is able to add these extended fields to an existing model as first class citizens.

Once the extended or subclassed fields are added, they can be treated like the existing fields or properties in being hydrated just like existing properties, added to a screen like existing properties, interacted via events like existing properties, and persited back to the underlying business view like existing properties.

There are several options available in Managed Code (C#, .Net Framework) that allows for what we are requiring to dynamically add properties to an existing model:

* Expando Objects
  + Difficulty in that it is X times slower than a dictionary
  + Difficulty serializing without added code
* Dynamic Objects
  + Difficulty is same as Expando objects
* Dictionary Objects
  + Difficulty in that there is no intellisense for the added properties

While the above three types of ways to dynamically extend the MVC models are viable, there is a major consideration stopping all of these from being a valid solution and that is the amount of code changes required to support this behavior.

Using one of the above approaches, when an MVC model is instantiated, the business view must be interrogated to discovered the extended fields. This is only done in the business repositories. However, an MVC model is instantiated in the controllers, and in the case of the “Create” or “Index” methods (i.e., no id specified or when specifying “New”), the controllers do not call down to the business repositories and therefore the model is not able to be extended in all places without hundreds or even thousands of code changes to the existing application.

Therefore, with this major obstacle, a different and less invasive approach was determined to add these extended fields to the MVC without the complexity and risk of making major changes to the application code.

The MVC models will be extended outside of the application, recompiled, and re-deployed. Therefore, at instantiation time of the MVC model, the extended properties will already be included in the model and thus no major modifications to the application.

Lets review the changes that are required in the application to support this approach.

ModelMapper.cs

The **ModelMapper** class is the base class for all MVC mappers in the web screens and performs two basic functions:

* Maps the data from the business view to the MVC Model when a GET request is made
* Maps the data from the MVC model back into the business view when a SAVE/UPDATE request is made

Since the model mapper for every model performs these two basic functions it is the perfect location to map these subclassed properties with no to minimal impact on existing web screens.

Map Getter

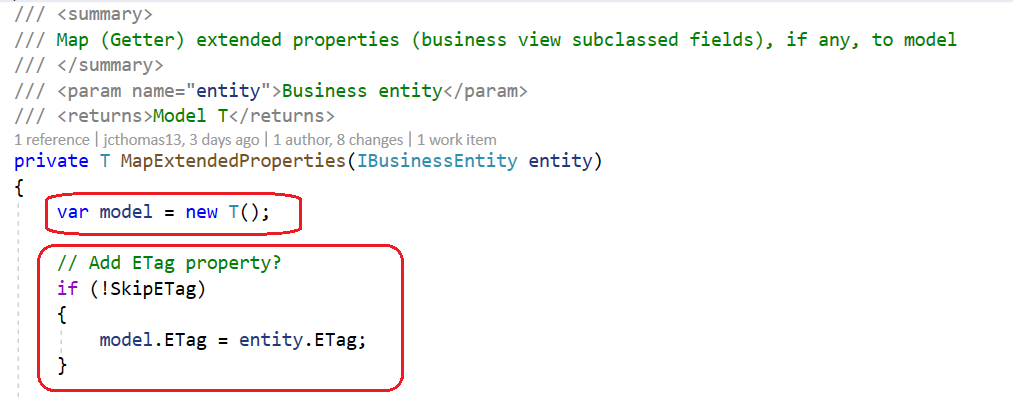
The GETTER routine (T Map) in the ModelMapper base class is where the model is instantiated and the ETag property is assigned from the business view to the model.

This is the new location for some framework code to discover and assign the extended properties in the business view to the extended properties in the MVC model.

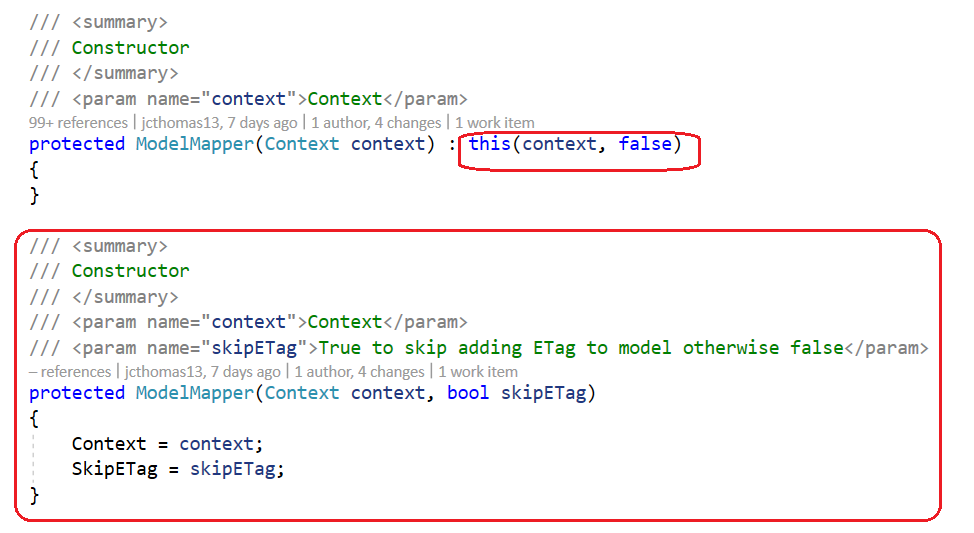
Here is a screenshot of a typical GETTER from AR Payment Codes that shows the function invoking the base and then mapping business view fields to the model:



Let’s see what is happening in the base class when the *base.Map(entity)* is executed:

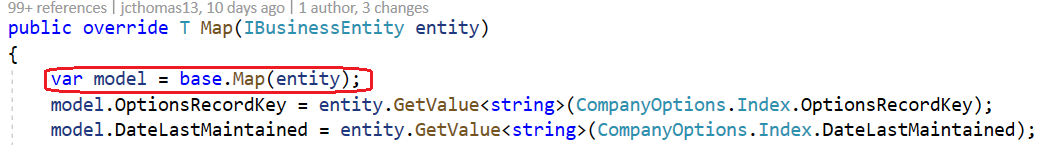


* The first thing it does is to instantiate the model. It was previously doing this as well.
* The next thing it does is conditionally assign the ETag property from the business view.
  + Previously, it always did this because when the *base.Map(entity)* was invoked in the screen’s mapper, the developer knew that the business view contained an ETag property
  + If the developer knew that the business view did NOT contain an ETag property, they simply did not call the base class and did the *new T()* themselves in the screen’s mapper
  + But with subclassing, the base class MUST be called since this is where the extended properties will be mapped since the framework will do this for the partner. Therefore, since all mappers now MUST invoke the base class, a parameter must be specified telling the base class to skip the ETag assignment.
  + This parameter has been added to the mapper’s constructor since modification of the existing signature would involve numerous changes to not only Sage’s code but partner code as well. The new constructor allows for the ETag flag to be specified:

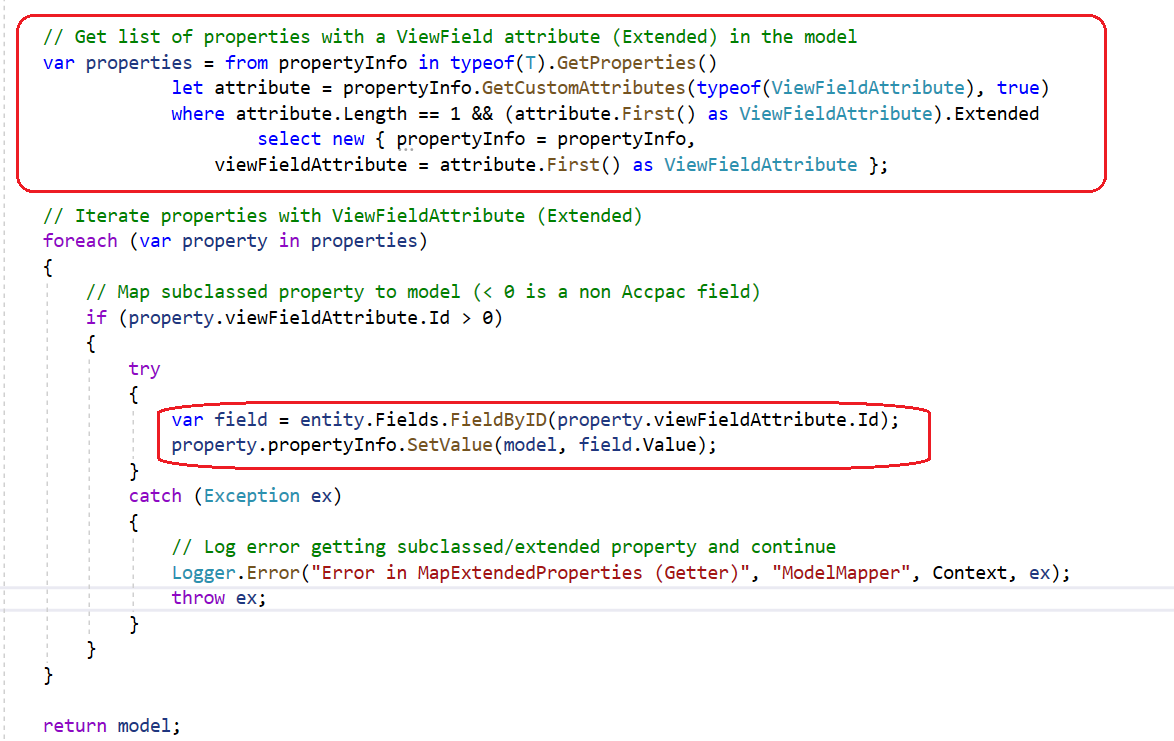


* + The first constructor is the base constructor that has always exists. It now invokes the new constructor with a default value of false for the SkipETag flag. Thus, all previous invocations of the base class constructor from the web screen’s mapper constructor have not changed.
  + The new constructor where the SkipETag property can be set is for the (~62) model mappers in the web screens that did NOT invoke the base class and did a new T() instead. By changing the screen’s model mapper instantiation in the contrsuctor of the screen’s repository to call the new base class constuctor, the screen’s GETTER (T Map) routine can now invoke the base class. Below is an example from the AP Company Options Mapper:





* + Now that we have covered what it previously did, let’s see what is new for the discovery and mapping of extended properties:

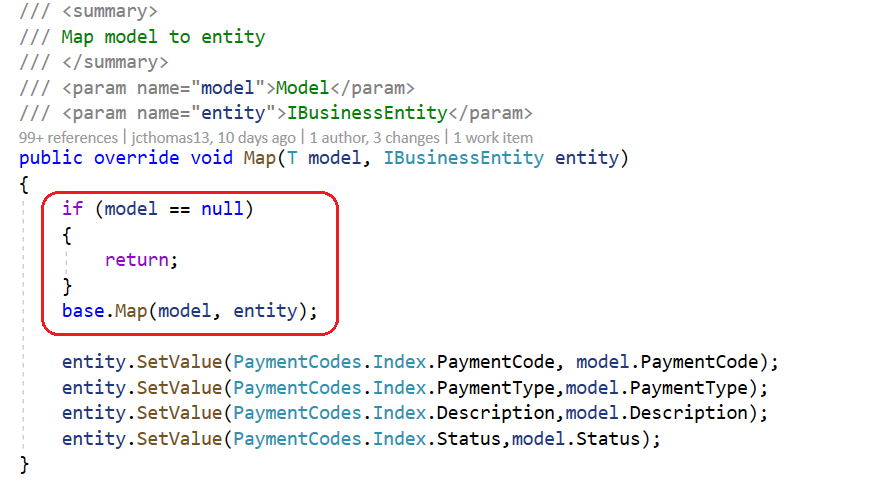


* + The first thing it does is uses reflection to get all of the MVC model’s properties that have the ViewField attribute with the Extended value to be true
    - The utility to add extended properties to the MVC model does this and will be discussed in later in this document
  + Any properties discovered to be extended will then be iterated and assigned from the business view to the model
  + Finally, the model with the already mapped extended properties is returned from the base class to the screen’s mapper for the mapping of business view properties to the model.
* In summary for the GETTER (T Map), since the signature did not change for the existing screens, the majority of the screen’s GETTER functions did not require any changes. The only changes required to the ~62 mappers were to add a parameter to the constructor and replace the new T() logic in the GETTER with a call to the base class (base.Map(entity)).

Map Setter

The SETTER routine (void Map (model, entity)) in the ModelMapper base class was previously marked as *abtract* and has been converted to *virtual* since this is the new location for some framework code to discover the extended properties in the business view and assign (map) then from the extended properties in the MVC model back into the extended properties in the business view.

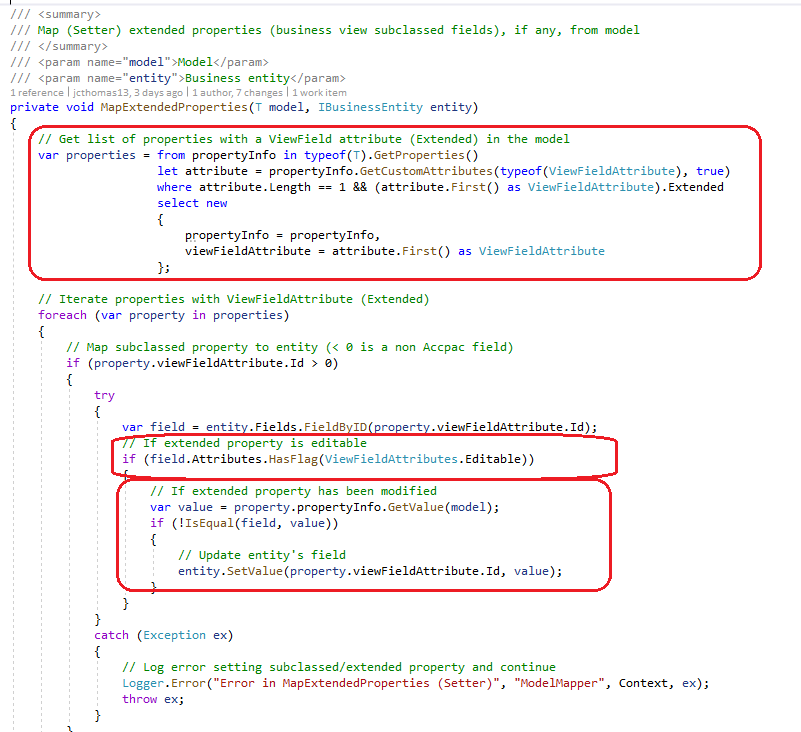
Here is a screenshot of a typical SETTER (void Map(model, entity)) from AR Payment Codes that shows the function invoking the new base method and then mapping business view fields from the model to the business view:



The defensive check to see if a model has any value is a good practice, yet only a small percetage of the screen’s SETTERs were doing it. It has been added to all screen SETTERs for consistency.

Since the base class SETTER was converted from abstract to virtual, framework code now exists in the base class to map extended properties in the model back into the business view. Therefore, all screen SETTERs were modified to invoke the base class method.

Let’s see what is happening in the base class when the base.Map(model, entity) is executed:



* + As noted, there was no code in the base class SETTER since it was previously an abstract method.
  + The first thing it does is uses reflection to get all of the MVC model’s properties that have the ViewField attribute with the Extended value to be true
    - The utility to add extended properties to the MVC model does this and will be discussed in later in this document
  + Any properties discovered to be extended will then be iterated and if the business view field is editable AND if the value has been changed since it was assigned to the model from the business view, it will be mapped from the model back into the business view.
  + Finally, the base class routine ends and returns to the screen’s mapper for the mapping of model properties to the business view properties.
* In summary for the SETTER (void Map(model, entity)), all mappers were modified to invoke the base class routine.

MapKey

* + No action at this time. Keys are established in the base classes by iterating the Fields collection of the entity and looking at the IsKey attribute. Since all fields, even sublassed fields are in this collection, no action is warranted.
  + However, all mapper classes have the Override for MapKey which the repositories can invoke directly. This is an override of the base.ModelMapper class that is abstract. We MAY need to change this to virtual and in every MapKey override in the screen’s mappers to do a base.MapKey(model, entity). However, it is not determined if this is needed at this time

BusinessEntityField and GridField

There is a class, BusinessEntityField, that inherits from ModelBase. This is fundamentally incorrect as this is a class that is used for FieldDefintions in the Business Entity. It was notcied that the BusinessEntity FieldDefinitions contained ModelBase behavior for every field. This is incorrect as the ModelBase is for an entire model and not a field.

I have removed the ModelBase inheritence from BusinessEntityField thus making this a lighter weight object and the ModelBase properties are not required.

This change also affected the GridColumns property in the ViewFinderViewModel class as it was Ienumerable<ModelBase> and it was changed to Ienumerable<GridField> since GridField which inherits from BusinessEntityField no longer inherits from ModelBase.

Customization

Overview

The Web customization strategy will be used to access the extended properties in the MVC model for the purpose of adding extended properties to the UI.

The beauty of adding the extended properties to the MVC model required no changes in the customization strategy since to the customization there is no difference between a Sage property and a partner property!

Sample

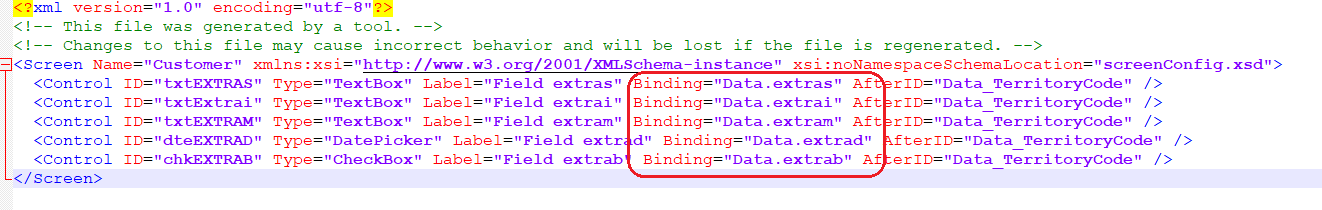
The following sample has extended/subclassed AR Customers as follows:

A screenshot of a computer

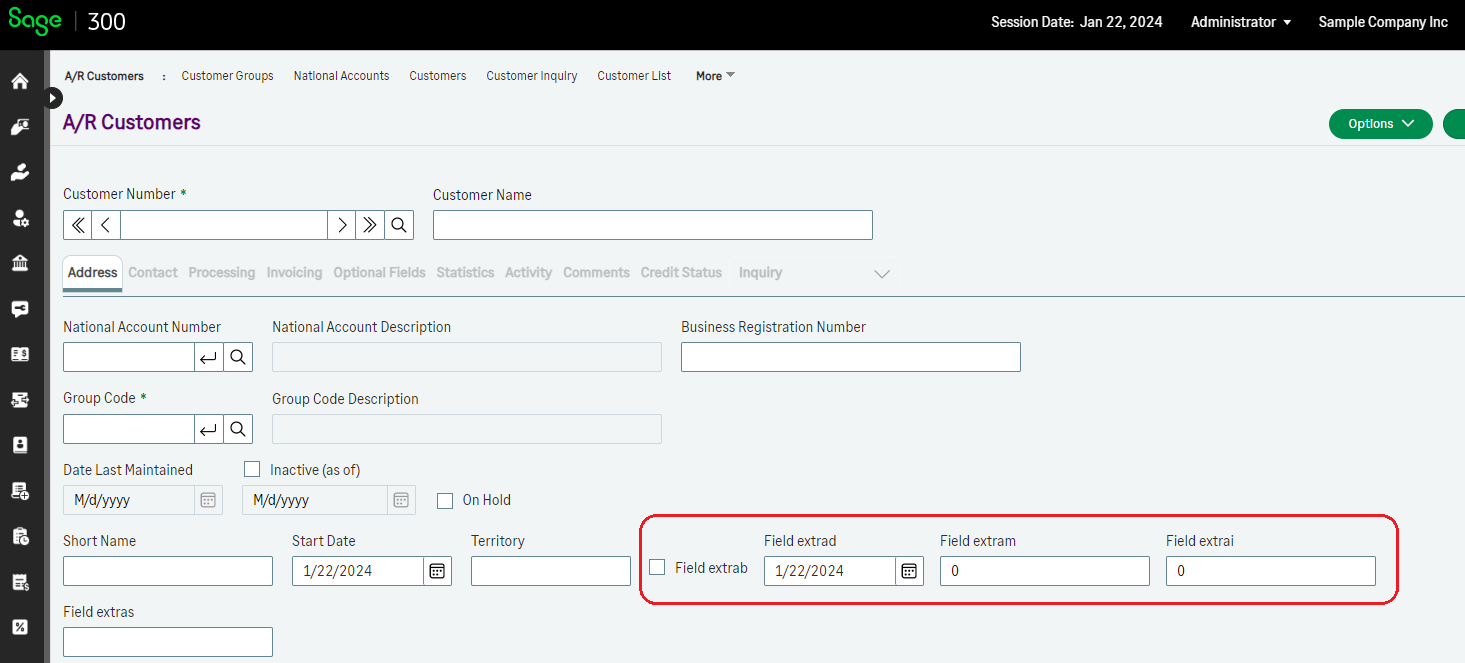
Description automatically generated

Five fields (EXTRAM, EXTRAD, EXTRAI, EXTRAB, EXTRAS) have been added to the ARCUS view for the purposes of testing the subclassing framework.

To support these fields being displayed on the AR Customer screen, the following customization (showing the XML only) was created to display the EXTRAS field:



When this customization is applied, the customer screen shows it:



Web API

Overview

The subclassing of business views to expose extended fields and functionalies is also required in the Web API. Where the Web API does not have a UI component, it does have a payload component. Thus where the Web Screens added extended fields to the UI through the customization strategy, the subclassed fields in the Web API must be added to the payload for the business view.

Strategy

The strategy for the Web API will follow a similar yet differnet approach. For the Web Screens, since only 1 screen can be displayed at a time AND since a model, let’s say Customers, is passed around in the application. However, this concept does not exist in the Web API since and endpoint, for let’s say ARCustomers, is not passed around. It is a destination. Therefore, a similar yet different approach was chosen.

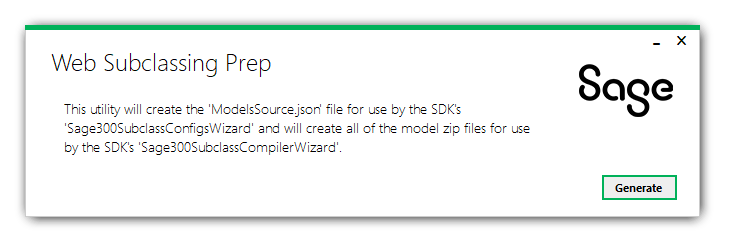
A partrner endpoint will be created that INHERITS from an existing endpoint. This stratgy is simple and straight-forward and also does not involve the customization strategy.

Utility and Wizards

Overview

An internal Sage utility, two new partner facing wizards, and a modification to the Code Generation Wizard will support the extended properties added by external partners to the MVC models in the web screens and a new partner facing Web API wizard will support extending endpoints in the Web API.

Web Subclassing Prep Utility



The Web Subclassing Prep is a Sage internal utility for generating 2 types of files to be used later in the partner facing wizards:

* JSON
  + A ModelsSource.json file will be created and later used in the Web Subclassing Configurations Wizard
  + This JSON file contains all of the models in the Sage web modules categorized by module. This will allow the configurations wizard to use this already populated list of modules and models instead of having to discover them.
* ZIP
  + Module zip files will contain the Visual Studio Models project for that module
  + These zip files will be used by the Web Subclassing Compiler and Deployment Wizard to use these projects for adding extended properties and later compilation and deployment.

Web Subclassing Configurations Wizard

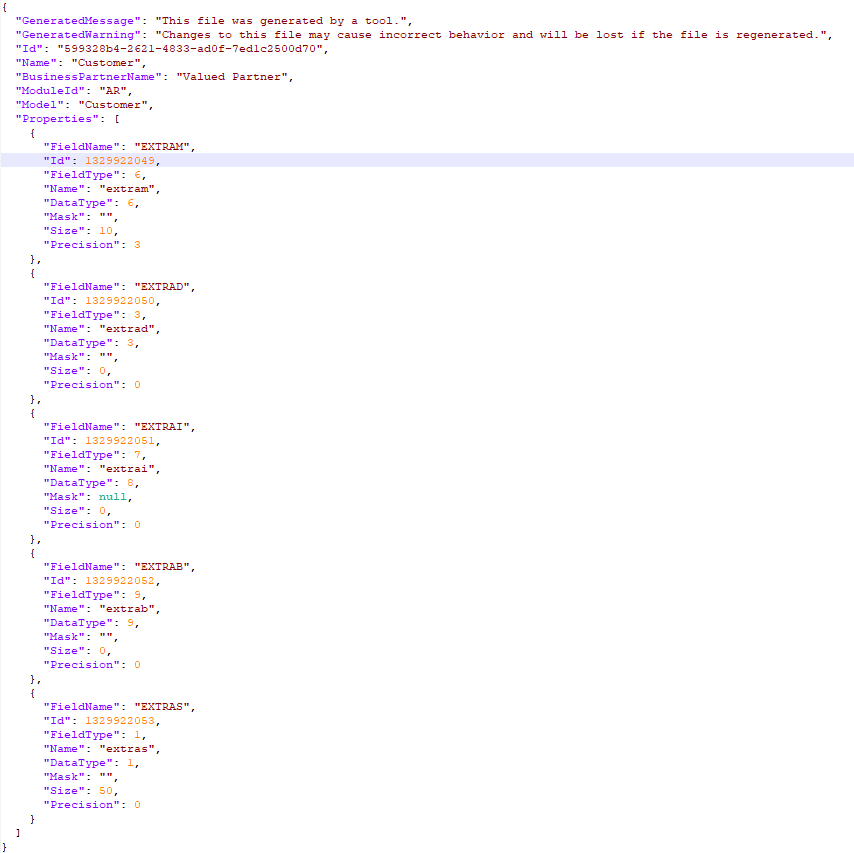
A screenshot of a computer

Description automatically generated

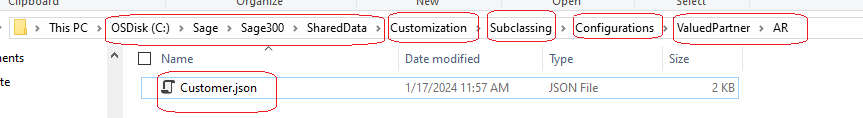
The Web Subclassing Configurations is a partner facing wizard for the generation and modification of subclassing configurations that will be later used by the Web Subclassing Compiler and Deployment Wizard.

In this wizard, the partner will identify what model in a particular module that has extended fields in the business view and then specify those extended properties. When the compiler wizard picks up these subclassed configurations the Sage model will be extended, recompiled, and deployed to the local installation whereby a customization can come along and display these extended properties in the UI.

The example used in this document to extended the AR Customer screen leveraged a configuration created by this utility to create that configuration. Here is the configuration:

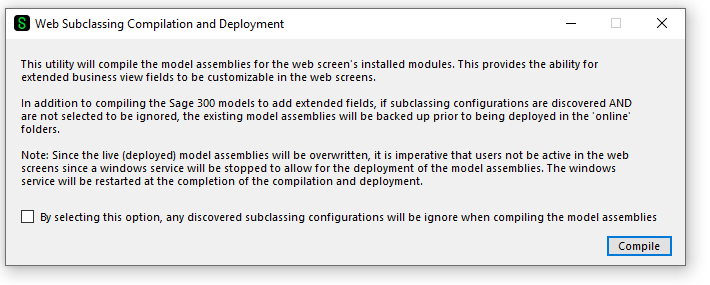


This will create the configurations in a known folder structure since this wizard MIGHT be moved from the Web SDK to the Sage 300 Admin Utilities start folder to allow multiple partners to apply subclassing configurations to the same model. However, there are certain challenges that must be addressed prior to this change. In the meantime, the folder location of the configurations on a partner machine are as follows:



* Shared data folder for web
  + Gathered from registry
* Customization
  + Known folder today where customizations are copied to upon import into web screens
* Subclassing
  + New folder for subclassing to store configurations
* Configurations
  + New folder sur subclassing to store configurations
* Partner folder
  + New folder based upon the value established in the configuration wizard. Allows for multiple partners. Future consideration if and when configuration wizard is moved to the application
* Module folder
  + New folder based upon the value established in the configuration wizard. Allows for categorization by module.
* \*.JSON
  + Configuration file name.

Web Subclassing Compilation and Deployment Wizard



The Web Subclassing Compilation and Deployment is a partner facing wizard for the consumption, or not, of subclassing configurations that will add extended properties to Sage 300 MVC models and then deploying the compiled assemblies to the local installation.

This wizard will determine which compiler to use and will prevent the wizard from continuing if no compiler is detected.

A bit now on compilers on a developer machine, and maybe eventually a client machine if this wizard is to be moved the application like the Configurations wizard:

* Visual Studio uses the *msbuild* utility for compiling and it can also be driven outside of the Visual Studio IDE.
* The .NET SDK uses the *dotnet build* utility (which is really a wrapper around the underlying msbuild utility) for compiling
* The .NET Framework delivers the *msbuild* utility and it was originally assumed that this utility could be used for all cases, but was soon discovered that the msbuild utility in the .NET Framework is ONLY there for compatibility and is only version 5.0. Our utility because of the .NET Framework version we are using (4.8) requires version 6+. Therefore, this *msbuild* utility is not an option.

Therefore, upon discovery, we will use the Visual Studio *msbuild* first if it is compatible otherwise we will use the .NET SDK *dotnet build* if it is compatible and abort the utility if no compatible compilers are found.

* As noted in the screenshot above, the partner also has the ability to regenerate the model assemblies AND to not include any configurations. I found this to be a particularly useful option when wanting to “reset” and remove any subclassed fields from the model assemblies.

So, what is the flow (conceptually) once the compilers are detected:

* The module zip files are copied down to and unzipped in the user’s temp + “Sage300Projects” folder and that’s where the compiler will pick up the csproj file and its files
* A backup folder will be created in the user’s temp + “Sage300Backups” + now() folder as this become the location to copy any existing model assemblies that will be overwritten in the ...\Online\Web\bin folder
* A build folder will be created in the user’s temp + “Sage300Build” folder as this is where the compiler will compile the model projects
* The projects will be iterated
  + If configurations exist for a module AND they have not been chosen to be excluded, the model’s class source file will be modified to include the specified prooperties in the configuration files. The below screenshot is an example of a configuration used in this document:

A screenshot of a computer

Description automatically generated

* + The model project will be compiled
  + The model assembly will be backed up from the ...\Online\Web\bin folder and copied to the backup folder. Note that only the ...\Online\Web\bin version of the assembly needs to be backed up since the other versions in the WebApi, Worker, etc. folders are the same copies.
  + The model assembly will be copied to the ...\Online\Web\bin folder and optionally to Worker, WebAPI, WebFR, etc. IF a version of the file to be copied exists in these other areas.
    - The worker service is also stopped on the first assembly copied and re-started when the compilation and deployment is complete. Therefore, running this wizard as “Administrator” ensure that valid permissons are given to the task to be able to stop and start the windows service.
* When the wizard is complete, before exiting the wizard will remove the source created to build the assemblies from the temp folder.
  + In the wizard’s build folder, this is where the partner will gather the modified model assemblies for deployment to their customers via THEIR subclassing package.
  + In the future, if this wizard were to be part of the application, the partner would need to simply deliver the configuration files since the client or customer (administrator) could compile and deploy as needed.

Web Subclassing Wizard

A screenshot of a computer

Description automatically generated

This wizard will support extending Sage 300 Web API endpoints where a new endpoint will inherit from an existing endpoint and the developer will add the extended fields.

This is accomplished by identifying the resource (module) and endpoint (model) to be extended.

The wizard will generate a Visual Studio solution and project where the developer will add the extended fields to the new endpoint which will then be discovered by the Web API when it starts.



# sage.com

# 0191 479 5911