Developer's Cookbook

Disassemble It

You can learn a lot about Active Commerce and Sitecore by utilizing a dissasembly tool. If you're interested in how something works, or doing an advanced customization/extension, this is the best way to inspect the product's behavior. Below are some options for such tools.

ILSpy	Redgate Reflector	JetBrains dotPeek

Registering Types in Microsoft Unity

Active Commerce utilizes the Microsoft Unity container to register implementations of various domain objects, services, and other types used throughout the product. This means that if you want to add new integrations or override other logic in Active Commerce, you can register overrides within Unity, either for your whole installation or for specific sites within Sitecore.

Active Commerce makes it easy to register your own types. At Sitecore startup, it will scan for any implementations of *ITypeRegistration* and *ISiteTypeRegistration* and execute them.

When registering your overrides in Unity, you need to be sure to configure all necessary properties and constructor parameters, and ensure you register using the proper lifetime manager. The best way to ensure this is to either follow the directions below, or reference *ActiveCommerce.IoC.RegisterTypes* in the *ActiveCommerce.Kernel* assembly using a tool such as Reflector, dotPeek, or ILSpy.

ITypeRegistration

By implementing *ITypeRegistration*, you can register types in the Unity container for the entire Sitecore / Active Commerce installation. The *SortOrder* property can be used to ensure the order in which types are registered. In most cases, it is safe to simply return a *0* value. See examples below on how to register specific types within Unity, or review *ActiveCommerce.IoC.RegisterTypes* in your reflector of choice.

```
public class RegisterTypes : ITypeRegistration
{
      public void Process(Microsoft.Practices.Unity.IUnityContainer
container)
```

```
	{
		//register your types here
	}

	public int SortOrder
	{
		get { return 0; }
	}
}
```

ISiteTypeRegistration

By implementing *ISiteTypeRegistration*, you can register types in Unity for specific sites within your Sitecore / Active Commerce installation. The *Sites* property can be used to return a collection of site names for which the registrations should apply. You can include the site names directly in your code, or perhaps reference the configured sites (*Sitecore.Sites.SiteContextFactory.Sites*) for those with a particular attribute configured (*SiteInfo.Properties*).

```
public class RegisterTypes : ISiteTypeRegistration
{
	public string[] Sites
	{
		get { //return string[] array }
	}

	public void Process(Microsoft.Practices.Unity.IUnityContainer
container)
	{
		//register your types here
	}

	public int SortOrder
	{
		get { return 0; }
	}
}
```

Adding a Custom Payment Provider

1. Create the Payment Provider class

- For an integrated/onsite payment provider (Authorize.Net, CyberSource), inherit from ActiveCommerce.Payment.IntegratedPaymentProviderBase.
    - Specify additional features that the provider will support by inheriting IReservable, and/or ICreditable.
- For an offsite payment provider (e.g. PayPal), inherit from ActiveCommerce.Payment.OnlinePaymentProviderBase.
    - Specify additional features that the provider will support by inheriting IReservable, and/or ICreditable.

- Implement the required method(s). Use the values from the method parameters as needed (PaymentSystem, PaymentArgs).
- Reference the Example Payment Provider on our GitHub.

2. Register with Unity

- Make sure to register it as a named instance (for our GitHub example, “MockService”).
- Reference the example registration on our GitHub.

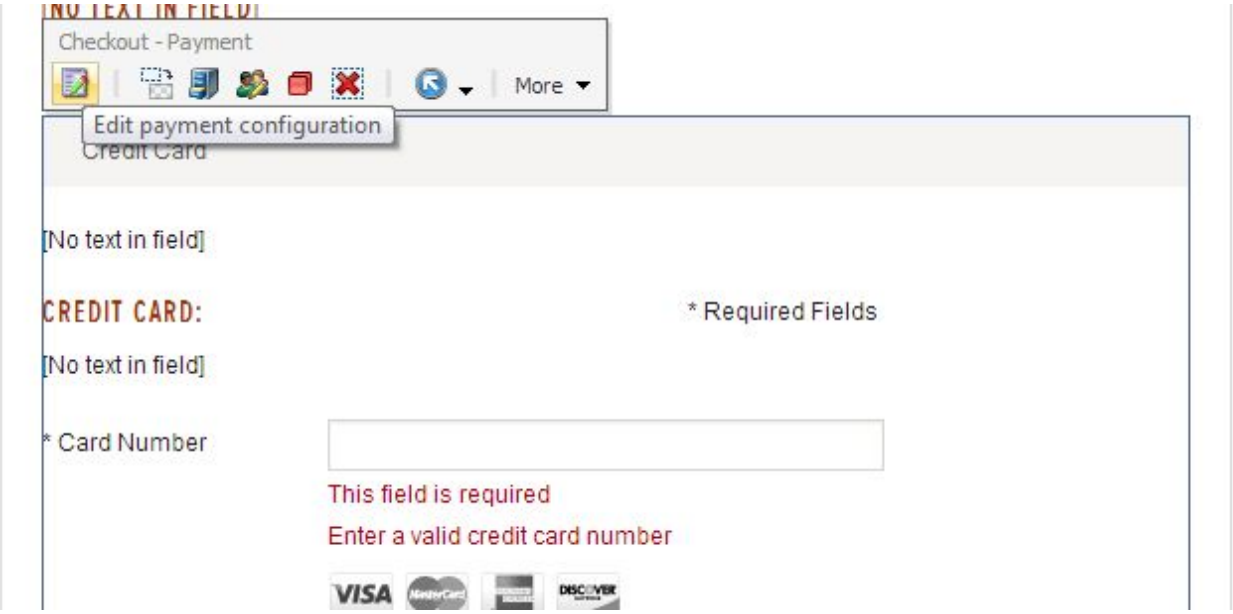
3. Add to Payment Options

- In Sitecore, add a new Payment to Webshop Business Settings/Payment Options
- Set the "Code" field to the same name you registered with Unity (e.g. "MockService").
- Those are the only fields technically required. The rest are there for you to use to make the provider configurable in Sitecore (e.g. *Username* and *Password* typically correspond to auth for your payment gateway, *Payment Provider Url* is typically the service endpoint for your payment gateway).
- The *Settings* field can be used for additional configuration settings. For example a gateway configured to do payment reservations might have settings such as:

```
<settings>
    <setting name="authorizeOnly">true</setting>
</settings>
```

4. Add to Checkout Page

- In Sitecore, go to the Checkout page, and open in Page Editor.
- For an integrated payment, you can switch the “Credit Card” payment component to use your new Payment Option.
    - Select the Credit Card *Checkout - Payment* component, and click the “Edit payment configuration” button



    - In the dialog, select your new Payment Option. Click OK and save.
- For an online payment provider (or other non-cc payment option), you can add another Checkout Payment component. Please refer to the *Active Commerce Configuration Guide - PayPal* section for an example of configuring a new payment option.
    - You can also reference the Invoice Payment Example on our GitHub.

Adding a Custom Shipping Service

1. Create the Shipping Service class

- Inherit from ActiveCommerce.Shipping.BaseShippingService or ActiveCommerce.Shipping.IShippingService
- BaseShippingService provides base implementations and helpers which may be useful (e.g. GetConfigurationSetting for accessing xml values in the “Service Configuration” field). If you don’t need any of this, simply use IShippingService.
- Implement the required properties and methods.
    - **Configuration** - Populated with xml from “Service Configuration” field
    - **ShoppingCart** - This will be populated with the current shopping cart instance
    - **Destination** - This will be populated with the customer shipping address
    - **ShippingWeight** - This will be populated with the cart total product weight
    - **Cachable** - Return boolean whether results of GetPrice should automatically be cached by Active Commerce (by Destination + ShippingWeight)
    - **Display** - Return boolean whether the current shipping option should be displayed to the customer
    - **GetPrice()** - Return the shipping price

- Reference the Shipping Service Example on our GitHub.

2. Register with Unity

- Make sure to register it as a named instance (e.g. "FedEx").
- Reference the example registration on our GitHub.

3. Add to Shipping Options

- In Sitecore, under Webshop Business Settings/Shipping Options, add a new item, or select an existing item.
- Set the “Service Code” field to the same name you registered with Unity.
- That is the only field technically required. You will more than likely have corresponding xml configuration to accompany your service, which would be entered in the “Service Configuration” field.

Configuring the Default Product URL Processor

Default Active Commerce product URLs take the format *ProductCode--ProductName*. Though the product name is part of the URL, it is not required that the name match the URL-encoded name of the product item in Sitecore. Only the product code needs to match. If you would like to change this behavior, or change the delimiter between the product code and name, you can re-register the URL processor in Unity, and inject different values for these properties of the processor. For example, the following registration would change the delimiter to two underscores and would force a name match.

```
container.RegisterType(typeof(Sitecore.Ecommerce.Catalogs.ProductUrlProcessor),
                        typeof(NameAndCodeAndVariantProductUrlProcessor),
                        "Item Name and Product Code",
                        new HierarchicalLifetimeManager(),
                        new InjectionMember[] {
    new InjectionConstructor(new
ResolvedParameter<Sitecore.Ecommerce.Search.ISearchProvider>(
        "CompositeSearchProvider")
    ),
    new InjectionProperty("ShopContext"),
    new InjectionProperty("CodeNameSeparator", "__"),
    new InjectionProperty("RequireNameMatch", true)
});
```

To create your own product URL processor to further customize product URLs, see the example on our GitHub.

Adding a Custom Tax Calculator

Active Commerce comes with 2 tax calculators. One for VAT (ActiveCommerce.Taxes.Vat.VatTaxCalculator), and one for North America (ActiveCommerce.Taxes.NorthAmerica.ShippingAddressTaxCalculator). The following instructions are for creating your own calculator.

1. Create the Tax Calculator class

- Inherit from ActiveCommerce.Taxes.TaxCalculatorBase or ActiveCommerce.Taxes.ITaxCalculator.
- TaxCalculatorBase provides access to the Sitecore fields related to this calculator via TaxConfiguration class. There are also base implementations and helpers which may be useful. If you don't need any of this, simply use ITaxCalculator.
- Implement the required properties and methods.
    - **TaxConfiguration** - This will automatically be set if using the TaxCalculatorBase. If using ITaxCalculator, you must define, but it's up to you if you'd like to use or not.
    - **Source** - This will be populated with the address of the company (as defined on /Webshop Business Settings/Company Master Data)
    - **Billing** - This will be populated with the customer billing address
    - **Shipping** - This will be populated with the customer shipping address
    - **Currency** - This will be populated with the current currency
    - **WillHandle()** - Return boolean whether this tax calculator will handle calculation for the addresses. If using TaxCalculatorBase, the MatchesLocation helper method is useful here if based on configured location (see step 3). If this will be your only tax calculator, then you could just return true.
    - **GetTaxes(IEnumerable<TaxInquiry> order)** - Return your TaxTotals
- Reference the Example Tax Calculator on our GitHub.

2. Register with Unity

- Make sure to register it as a named instance (e.g. "MyTaxCalc").
- Reference the example registration on our GitHub.

3. Add to Tax Calculations

- In Sitecore, under Webshop Business Settings/Tax Calculation, add a new item. Insert from Template, and choose /sitecore/templates/ActiveCommerce/Business Catalog/Tax Calculator
- Set the "Tax Calculator" field to the same name you registered with Unity (e.g.

"MyTaxCalc").
- That is the only field technically required. You can use the Locations field if you'd like to restrict the use of this calculator to specific countries/regions (in conjunction with the MatchesLocation helper method). You can also extend this template with your own fields, and create a new TaxConfiguration to access those values.

Adding a Custom Tax Rate Provider

The built-in North America tax calculator makes use of a Rate Provider. See the *Active Commerce Configuration Guide* for a list of Rate Provider plugins already offered. The following instructions are for creating your own provider.

1. Create the Tax Rate Provider class

- Inherit from ActiveCommerce.Taxes.Rates.ISalesTaxRateProvider.
- Implement the required method(s).
- Reference the Example Rate Provider on our GitHub.

2. Register with Unity

- Make sure to register it as a named instance (e.g. "training" or "MyRateProvider").
- Reference the example registration on our GitHub.

3. Update Tax Calculations to use

- Follow the Active Commerce Configuration Guide - Rate Provider section, except we'll set the "Rate Provider" field to the name you registered with Unity (e.g. "training" or "MyRateProvider").

Adding a Custom Address Validator

Active Commerce includes a basic server-side address validator that confirms address format for U.S. and Canadian addresses. The Fast Tax plugin includes an address validator that uses the Fast Tax rate API to confirm that tax rates can be resolved for the given address. Some validation is also done client side via AngularJs validation directives. If you wish to implement custom server-side validation rules, or utilize another address validation API, you can implement a custom address validator. Address validators can also be used to make changes to the provided address. Interactive address validation (e.g. having the user confirm the corrected address) currently requires customizing the checkout experience.

1. Create the Address Validator class

- Inherit from ActiveCommerce.Validation.IAddressValidator.
- Implement the required method(s).

- Reference the Example Address Validator on our GitHub.

2. Register with Unity

- Reference the example registration on our GitHub.

Adding a Product Detail Page Tab

1. Create a user control

- Inherit from AbstractGlassUserControl, ActiveCommerce.Web.skins.IProductDetailsControl and ActiveCommerce.Web.skins.ITabControl
- Implement the required properties.
    - "Model" and "ViewModel" will be set automatically and available for use in your control
    - "TitleKey" should return a key corresponding to a value in the Translation Dictionary (Add a new Dictionary entry if necessary)
    - "Id" should return a unique (DOM) identifier (this will be the id of the tab element)
- Reference the example sublayout usercontrol and its code-behind on our GitHub.

2. Create a sublayout in Sitecore

- Set your user control as the Ascx file

3. Add sublayout to presentation details

- Add your sublayout to the /sitecore/templates/ActiveCommerce/Product/Product/__Standard Values presentation details -- or, whichever product template this tab should appear on e.g. one of your product templates.
    - If you wish to apply to all your product types, it's better to create your own "base" template for products and add the rendering there.
- Use sorting to define where your tab appears (look for "Product Tab - ", these are the default Active Commerce tabs)

Order Processing Pipelines

If you wish to extend Active Commerce order processing, it's helpful to understand the order

processing pipelines, their existing processors, and how the pipelines interact. Order processing always starts with the *acStartOrderProcessing* pipeline.

acStartOrderProcessing

This pipeline is invoked at the start of order processing. When invoked, the *ShoppingCart* and *CheckOut* properties are populated from session on the *OrderPipelineArgs*. From there, the steps execute as follows:

CreateOrUpdateAccount	Based on the user login state and choices made during checkout, the user may need a new account created. If the current user account does not have a saved shipping/billing address, the addresses input during checkout will be saved to the user's account. If you wish for the checkout process to always replace the saved shipping/billing address, set the *AlwaysUpdateExisting* property on the processor to "true."
GenerateOrderNumber	Creates a new order number using the *OrderIDGenerator* registered in Unity, which by default is the *HiLoOrderIdGenerator*. This generator uses NHibernate's HiLo ID generation to create a new order number. The order number is populated in the *OrderPipelineArgs* and in the shopping cart. Read an explanation of how HiLo identity generation works.
CreateOrder	Translates data from the shopping cart into the order domain model and populates the *Order* property on the *OrderPipelineArgs*. The specific types that are created are determined by the currently registered *IOrderFactory* implementation. If you wish to extend the *Order* class or related entities, you'll need to extend the order factory as well. Note that the order is not yet persisted when this step completes.
VerifyOrder	Ensures that an order was properly created and populated on the *OrderPipelineArgs*.
SaveOrder	Persists the order to the database using the *IOrderManager* registered in Unity. If you extend the order domain model and wish to save additional data about the order, patch your new processor between *VerifyOrder* and *SaveOrder*.
StartPayment	Resolves a payment provider based on the payment

	system configured on the shopping cart, and populates the *PaymentProvider* property on the *OrderPipelineArgs*. Invokes the payment provider. If the payment provider is an integrated, on-site payment gateway (*OnlinePaymentProvider*), the pipeline continues. If the payment provider is an off-site payment gateway which will require an HTTP redirect (e.g. Paypal), the processor first ensures that nothing went wrong with the initial communication with the gateway, then terminates the pipeline. The user at this point will be redirected to the gateway's site, and upon return the *acResumeOrderProcessing* pipeline will be invoked.
CompleteOrderProcessing	If the pipeline is still executing at this point (i.e. for onsite payment methods), the *acCompleteOrderProcessing* pipeline is invoked.

acResumeOrderProcessing

This pipeline is (indirectly) invoked by the *Checkout-PaymentReturn* rendering on the payment return page (*/shop/checkout/return*), which should be configured as the return page in your payment gateway. Its purpose is to "rehydrate" the shopping cart and order, and process the data returned by the offsite payment gateway. This pipeline is not invoked for onsite/integrated payment gateways.

Resuming order processing is currently dependent on the shopping cart still being present in the user session.

FindOrder	Using the order number on the shopping cart, retrieves the in-process order from the database and populates the *Order* property on the *OrderPipelineArgs*.
FindShoppingCart	Ensures that the shopping cart is still in session and populates the *ShoppingCart* property on the *OrderPipelineArgs*.
ProcessPaymentCallback	Resolves a payment provider based on the payment system configured on the shopping cart, and populates the *PaymentProvider* property on the *OrderPipelineArgs*. Invokes *ProcessCallback* on the payment provider, which should determine the status of the payment based on data posted by the offsite payment provider.
CompleteOrderProcessing	Invokes the *acCompleteOrderProcessing* pipeline.

acCompleteOrderProcessing

This pipeline is invoked at the completion of either the *acStartOrderProcessing* or *acResumeOrderProcessing* pipeline, depending on the type of payment being used by the customer (onsite or offsite).

VerifyPayment	Checks the status of the payment and persists the payment details to the order. If you are implementing a custom payment method, you may wish to extend this processor to record additional payment details to the order record. Note that even if a payment fails or is canceled, the order will remain in the orders database, with an appropriate order status. This is helpful for debugging order failures. If payment fails or is canceled however, the pipeline aborts following this step.
SetPipelineSuccess	This processor sets the *Status* property on the *OrderPipelineArgs* to *Succeeded*. All essential operations for order processing should be complete by this point. If you wish to validate and/or integrate order data with an external system, and that call is essential to your order processing, inject your processor between *VerifyPayment* and *SetPipelineStatus*.
OpenOrder	Sets the order status to *Open*, indicating that processing is complete. Sets order sub-state based on payment status (authorized or capture).
SendEmailToCustomer	Invokes a Sitecore job which sends an email receipt to the customer based on the *Order Mail To Customer* email template.
SendEmailToAdmin	Invokes a Sitecore job which sends an email notification to the site *Sales* contact based on the *Order Mail To Admin* email template. If you wish for a sales contact to receive this email, be sure to configure the *Sales* contact email under *Company Master Data*. If you do not wish for an email to be sent to a sales contact for every order, you should utilize a configuration patch to disable this processor.
DecrementStock	Decrements stock for all purchased products by the purchased quantity. If your product catalog is entirely virtual/digital, or if you are not tracking or caching stock in Active Commerce, you should utilize a configuration

	patch to disable this processor.
ClearCart	Clears out all shopping cart lines, shipping provider, discounts, coupon codes, and payment information on the shopping cart.
RecordEngagementValue	Adds the total order purchase value to the Sitecore Engagement Value of the current visit/interaction. This is useful if you are not otherwise using Engagement Value to score visit quality. However if you are otherwise applying and tracking a score on visit goals, etc., you will likely want to utilize a configuration patch to disable this processor, as it may skew your analytics.

acOrderProcessingError

This pipeline is invoked on any error during order processing. Each step can be configured with a *Threshold* corresponding to a value in the *OrderProcessingStatus* enumeration, above which the processor will not execute. This pipeline attempts to capture and convey as much information about the order as possible to assist in debugging. If you have another error logging or notification framework you are utilizing (e.g. ELMAH or similar), you may wish to add a processor to this pipeline to report errors with order processing. Note that additional errors encountered while executing this pipeline should accumulate and report with the original order error.

MarkOrderAsFailed	Sets the order state in the order database as “Failed.”
ReversePayment	If a payment was successfully completed prior to a fatal order processing error, this processor attempts to either cancel a payment authorization, or credit a payment capture, as necessary.
LogError	Logs detailed error and order debugging information to the Sitecore logs.
SendErrorEmail (Sales)	Sends an email error report to the sales contact configured under *Company Master Data*. Be sure to configure this email address if you wish for a sales contact to receive the error email.
SendErrorEmail (Technical)	Sends an email error report to the technical contact configured under *Company Master Data*. Be sure to configure this email address if you wish for a technical contact to receive the error email.

Order Domain Model

The order domain model originates in Sitecore E-commerce Services, which based the model on an OASIS standard known as “Universal Business Language.” The model is rather robust and thus it can be somewhat difficult/confusing to find particular fields or values within the model.

The API hints below represent the object path to some commonly used values.

Customer Email
`order.BuyerCustomerParty.Party.Contact.ElectronicMail`

Customer’s Full Name
`order.BuyerCustomerParty.Party.PartyName`

Billing Address
`order.BuyerCustomerParty.Party.PostalAddress`

Billing First Name
`order.BuyerCustomerParty.Party.Person.FirstName`

Billing Last Name
`order.BuyerCustomerParty.Party.Person.FamilyName`

Billing Phone
`order.BuyerCustomerParty.Party.Contact.Telephone`

Shipping Address
`order.DefaultDelivery.DeliveryParty.PostalAddress`

Shipping First Name
`order.DefaultDelivery.DeliveryParty.Person.FirstName`

Shipping Last Name
`order.DefaultDelivery.DeliveryParty.Person.FamilyName`

Shipping Phone
`order.DefaultDelivery.DeliveryParty.Contact.Telephone`

Order Lines
`order.OrderLines`

Order Line Product Code
`line.LineItem.Item.Code`

Order Line Price
`line.LineItem.Price.PriceAmount.Value`

Order Line Quantity
`line.LineItem.Quantity`

Order Total, Including Tax
`order.AnticipatedMonetaryTotal.TaxInclusiveAmount.Value`

Order Tax Total
`order.TaxTotal.TaxAmount.Value`

Address Line
`address.AddressLine`

Address City
`address.CityName`

Address State/Province
`address.CountrySubentity`

Address Country Code
`address.CountryCode`

Address Postal Code
`address.PostalZone`

Adding an Order Pipeline Processor

1. Create an order processor class

- Inherit from ActiveCommerce.Orders.Pipelines.OrderPipelineProcessor
- Implement the required “DoProcess” method
    - Use OrderPipelineArgs argument as necessary
        - ShoppingCart contains the state of the Shopping Cart when ordered
        - Order contains the order information (if after the “CreateOrder” step)
        - To abort the order process, set the *Status* and then *AbortPipeline()*
            - Optionally call *AddMessage* to change failure message displayed to customer
- For integrating order data externally, reference the Order Integration Example.
- For adding data to persist to the order, reference the Gift Message Example.
- For extending payment data saved with the order, reference the Invoice Payment Example.

2. Inject the processor into the appropriate order processing pipeline

- Utilize a Sitecore configuration patch to add the processor to the appropriate location in the appropriate pipeline.
    - If integrating order data externally, add your processor to the *acCompleteOrderProcessing* pipeline between the *VerifyPayment* and *SetPipelineSuccess* steps.
        - Example
    - If adding data to be persisted with the order, add your processor to the *acStartOrderProcessing* pipeline between the *CreateOrder* and *VerifyOrder* steps.
        - Example
    - If extending payment data saved with the order, replace the *VerifyPayment* step of the *acCompleteOrderProcessing* pipeline.
        - Example

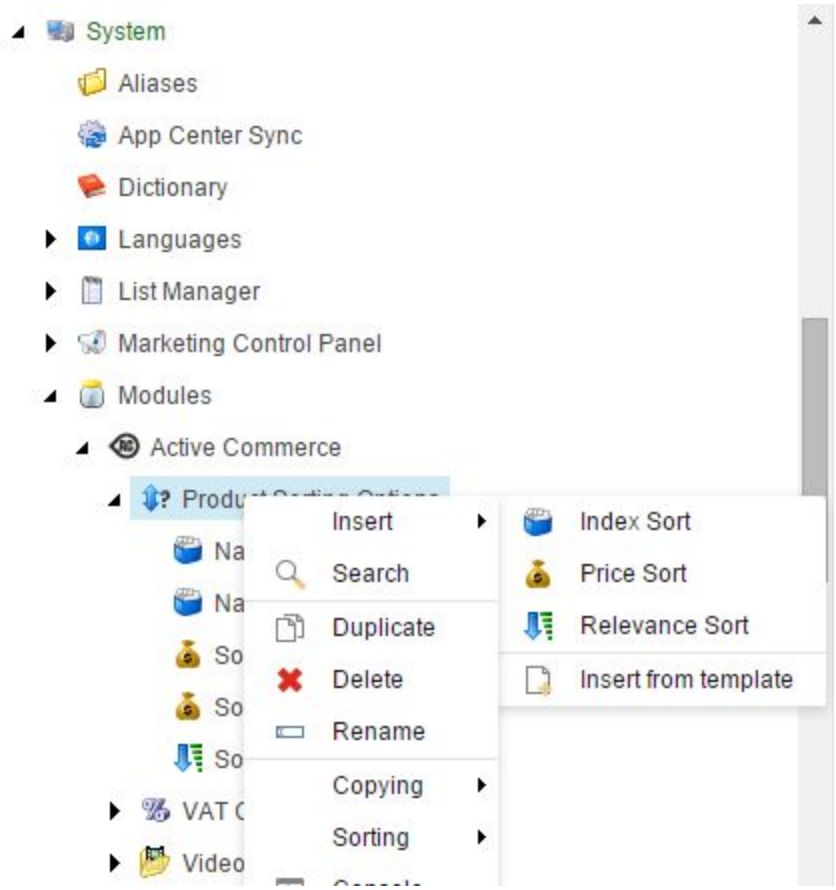
Adding Product Sorting Options

Product sort items can be added to provide additional sorting options on product catalog categories. *See the Active Commerce User Guide for assigning sort options to catalog categories.*

Index Field Sort

An index field sort is built in to Active Commerce and sorts on fields available in the search index. No additional coding is required, only Sitecore configuration.

- In Sitecore, navigate to Sitecore/System/Modules/Active Commerce/Product Sorting Options



- Insert a new Index Sort as shown above.
- Set the Display Value to the text you want to see on the Category Product List page; this is the text the user will select when performing a sort.
- Check the Reverse Sort box if this should sort in reverse order.
- In the Index Field field, add the name of the field from the search index to perform the sort on. For example, in a default Lucene index, you might add "_name" to sort by product name.
- If needed, repeat this process for adding a second option to sort in the reverse order of the first option.
- Move each option created through workflow and publish.
- New options will now be available to use on category pages. See the *Active Commerce User Guide* for details on assigning sort options to catalog categories.

Dynamic Product Sort

A dynamic product sort can apply business rules or draw on external data. While they don't actually perform the sorting, they provide a delegate function that gives a value to sort on.

Create a new template

- Inherit from */sitecore/templates/ActiveCommerce/Product Sorting/Product Sort Base* and

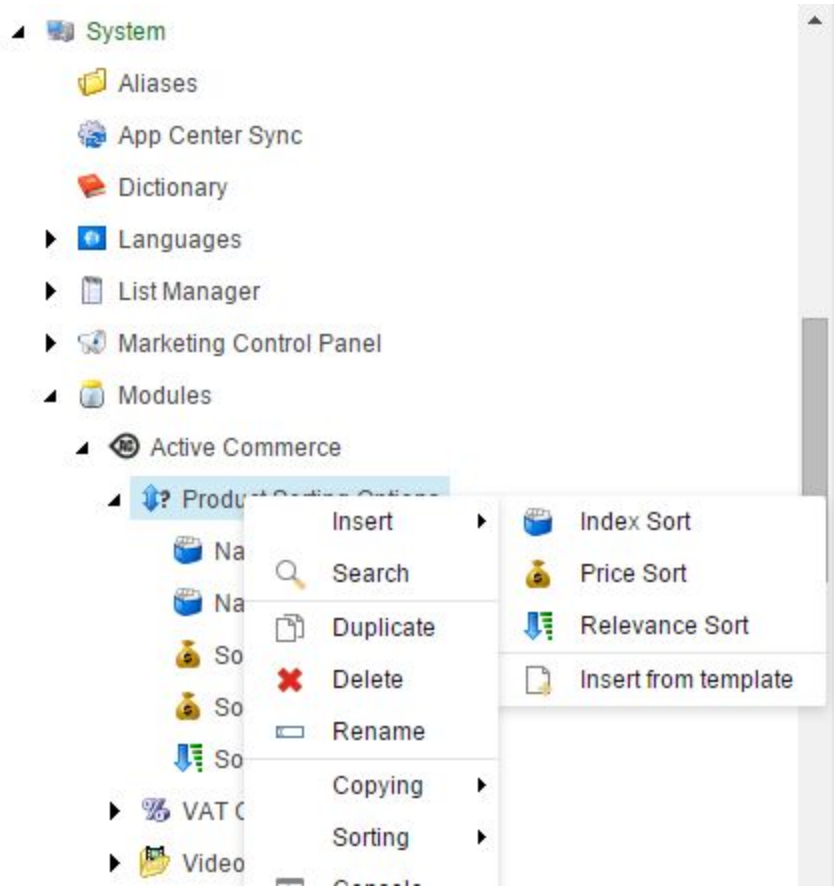
optionally, from */sitecore/templates/ActiveCommerce/Product Sorting/Reversible Sort Base* if you'd like your sort to be reversible.
- Note the template's ID

Create a new class

- Inherit from ActiveCommerce.Products.Sorting.IProductSort
- Add a *SitecoreType* Glass attribute, setting the "TemplateId" to the ID of the template created above
- Implement the required properties. These should be mapped using Glass attributes (see example below). If you inherited *Reversible Sort Base*, you'll also want to map in the "Reverse Sort" field.
- Implement the required "PopulateSearchOptions" method. This method should populate the following properties of the passed in RepositorySearchOptions:
    - SortFunc - supplies the value to sort on during dynamic sort with a delegate of type Func<ActiveCommerce.Products.ProductBaseData, object>
    - ReverseSort - (optional) set this to your mapped in "Reverse Sort" field
- Reference the Sort by Rating example on our GitHub.
- You can also inspect the included price sort (ActiveCommerce.Products.Sorting.PriceSort) utilizing a dissasembly tool. Notice that in this one, the SortFunc differs based on the ReverseSort value (we use min price when sorting ascending, max price when descending).

Add Sort Option(s) to Sitecore

- In Sitecore, navigate to Sitecore/System/Modules/Active Commerce/Product Sorting Options



- Using “Insert from template”, add a new item selecting your template. Alternatively, you can configure Insert Options for the “Product Sorting Options” item.
- Set the Display Value to the text you want to see on the Category Product List page; this is the text the user will select when performing a sort.
- Check the Reverse Sort box if this should sort in reverse order.
- If needed, repeat this process for adding a second option to sort in the reverse order of the first option.
- Move each option created through workflow and publish.
- New options will now be available to use on category pages. See the *Active Commerce User Guide* for details on assigning sort options to catalog categories.

Customizing the Checkout

Prerequisites

- Familiarity with AngularJs - a Javascript framework from Google for creating dynamic web apps. This is used heavily by the checkout. The main website is the best place to start. There are a ton of other great resources out there as well, such as https://egghead.io/.

- Using and developing for the Sitecore Page Editor

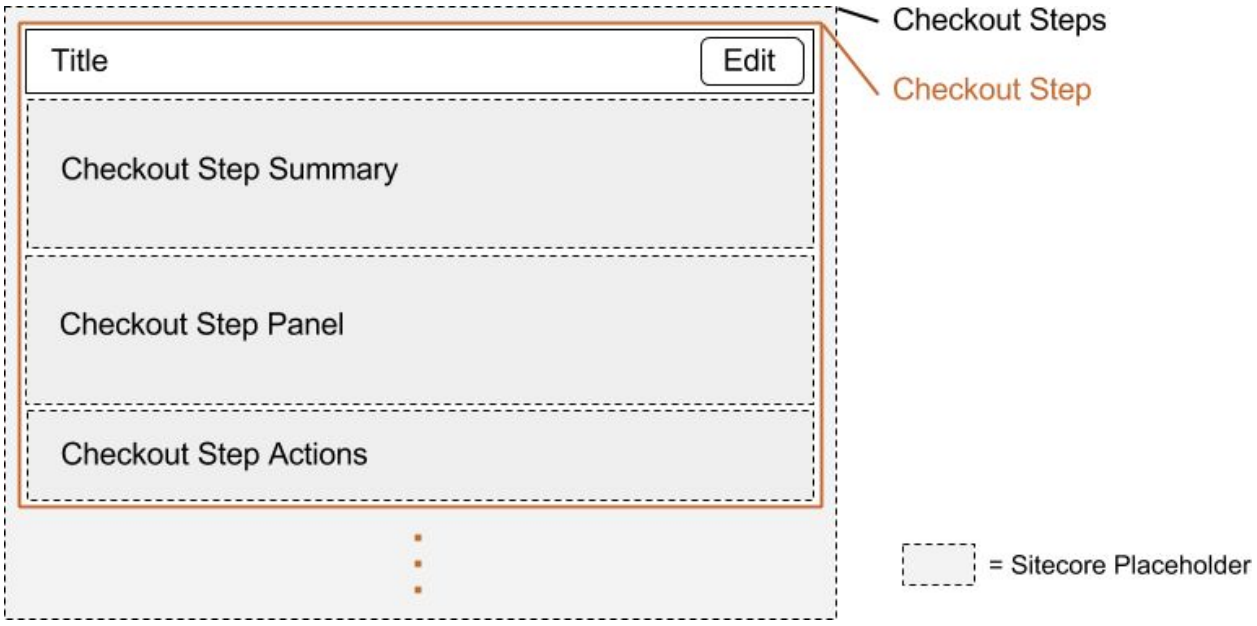
Architecture

The Active Commerce checkout was developed using a modular, component-based approach to allow for easy extension. In general, the checkout is built using standard Sitecore best-practices - placeholders, sublayouts, and backing datasource items. This allows full management of the page structure using the Sitecore Page Editor, as well as A/B testing. However, once the page is rendered, AngularJs takes over, which gives the checkout its dynamic “single page app” nature.

There are 3 main structural pieces involved: **Steps**, **Components**, and **Actions**.

Steps

Steps represent the checkout steps that are visible to the customer. e.g. Shipping, Payment, Review. Each step (“Checkout Step”) consists of a Title, Edit button, and placeholders - *Checkout Step Summary, Checkout Step Panel*, and *Checkout Step Actions* .



Placeholders

There are 3 main placeholders involved:

- *Checkout Step Summary* - This section will display once the step is *complete*. Allowed controls: Checkout Components.
- *Checkout Step Panel* - This section will display when the step is *active*. Allowed controls: Checkout Components.
- *Checkout Step Actions* - This section will display when the step is *active*. Allowed controls: Checkout Actions.

You'll also notice the *Checkout Steps* placeholder in the diagram above. This is the placeholder to which all Checkout Steps are added.

States

An individual step will go through different states throughout the checkout process. Only 1 step will be current at a time; this is the *active* step. Once a step has passed, the step is considered *complete*. A step is also considered *editable* (all are editable by default).

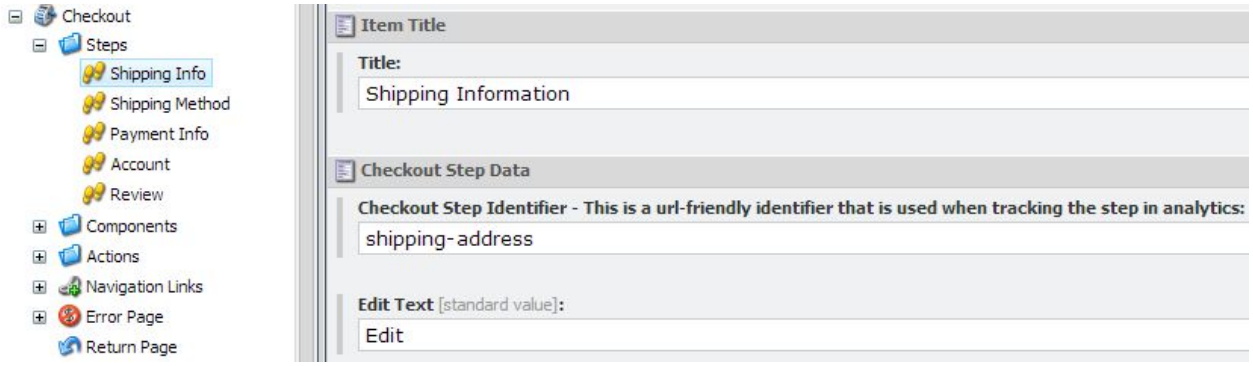
These states affect the Checkout Step accordingly:

- The Edit button - The edit button will display once the step is *complete* and the step is also *editable*.
- The placeholders visibility - See *Placeholders*
- CSS - Classes (i.e. "active", "complete", "editable") are added to the checkout step container when in a given state

Datasource Item

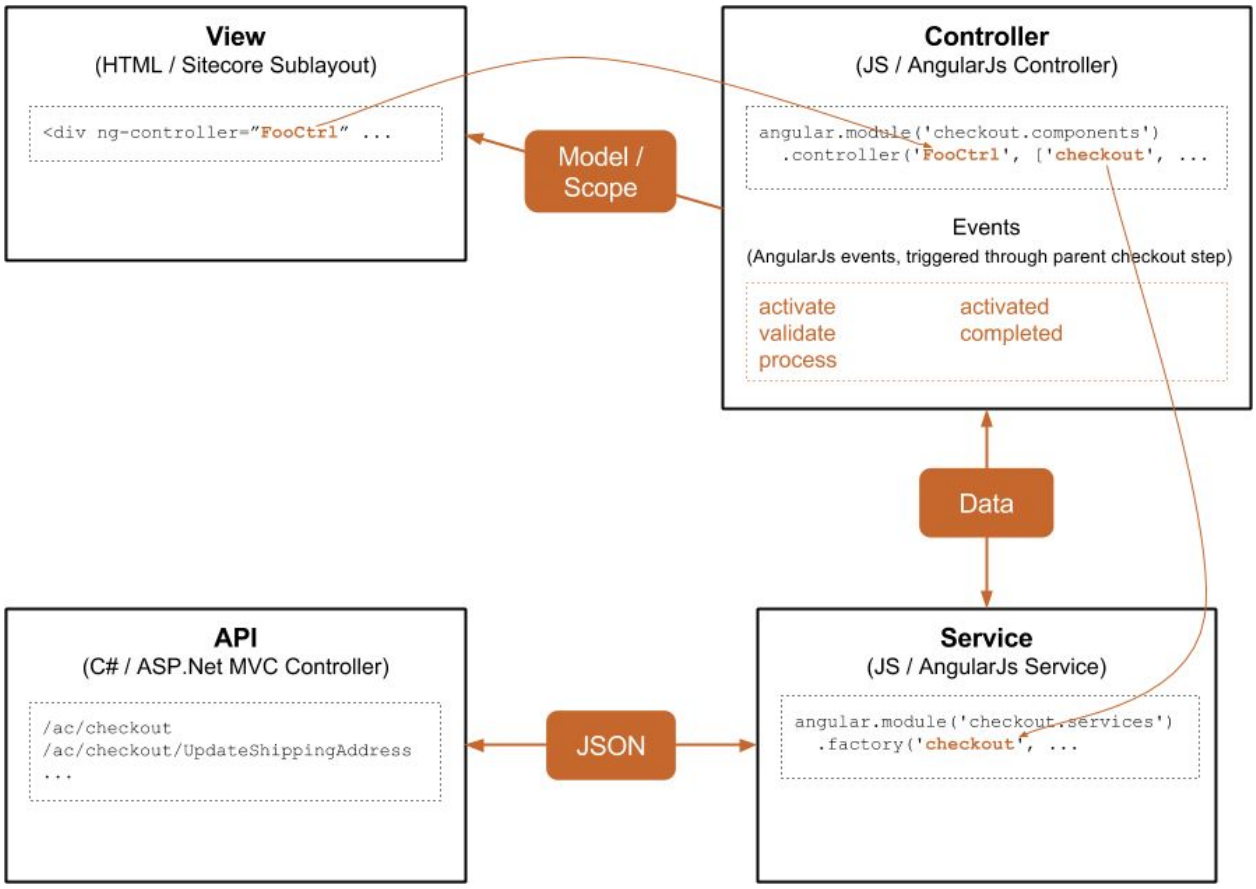
Steps use the *ActiveCommerce/Purchasing/CheckoutComponents/Checkout Step* template. Note the "Checkout Step Identifier" field. This is used to uniquely identify the step in analytics, and also in javascript.

An example corresponding datasource item in Sitecore:



Components

Components are added to the *Checkout Step Summary* and *Checkout Step Panel* placeholders. It is the components which perform all business logic and communicate with the server. Here is an overview of the pieces involved:



Each component is comprised of at least a **View** (Sitecore sublayout) and a corresponding **Controller** (AngularJS). There may also be additional **Services** and **API** endpoints required.

In general, the View, Controller and Services are all vanilla AngularJs, following best-practices.

- The View has a `ng-controller` attribute, which tells AngularJs which controller should be used. This gives the View its model/scope.
- Through AngularJs dependency injection, the Controller declares the Services it needs access to. In this case, the `checkout` service.
- Any requests to the server are done in a Service.
- Services use AngularJs `$http` (`get, post`) to make Ajax requests to the API.
- ASP.Net MVC controllers respond to requests, served up as JSON.

Events

Events are what tie the Checkout Steps to the nested Components. As a customer progresses through the checkout, AngularJs events are broadcast at the Checkout Step level, which, by nature of AngularJs eventing, will reach all nested components. It is up to the component's controller to handle these events and act and/or respond accordingly.

There are 2 types of events - *deferred* and *non-deferred*.

Deferred events allow the listener (Component) to queue up additional work to be performed. The caller (Checkout Step) will not proceed until all queued up work is completed. Deferred events can also be rejected. If a listener rejects the event, the caller will not proceed at all. These are the built-in deferred events:

- *activate* - triggered when the Checkout Step is about to be activated
- *validate* - triggered when user clicked on Next (see Actions below)
- *process* - triggered when user clicked on Next, after a successful *validate* event

Non-deferred events do not allow the listener any control over the outcome of the event. They are essentially “FYI only”. These are the built-in non-deferred events:

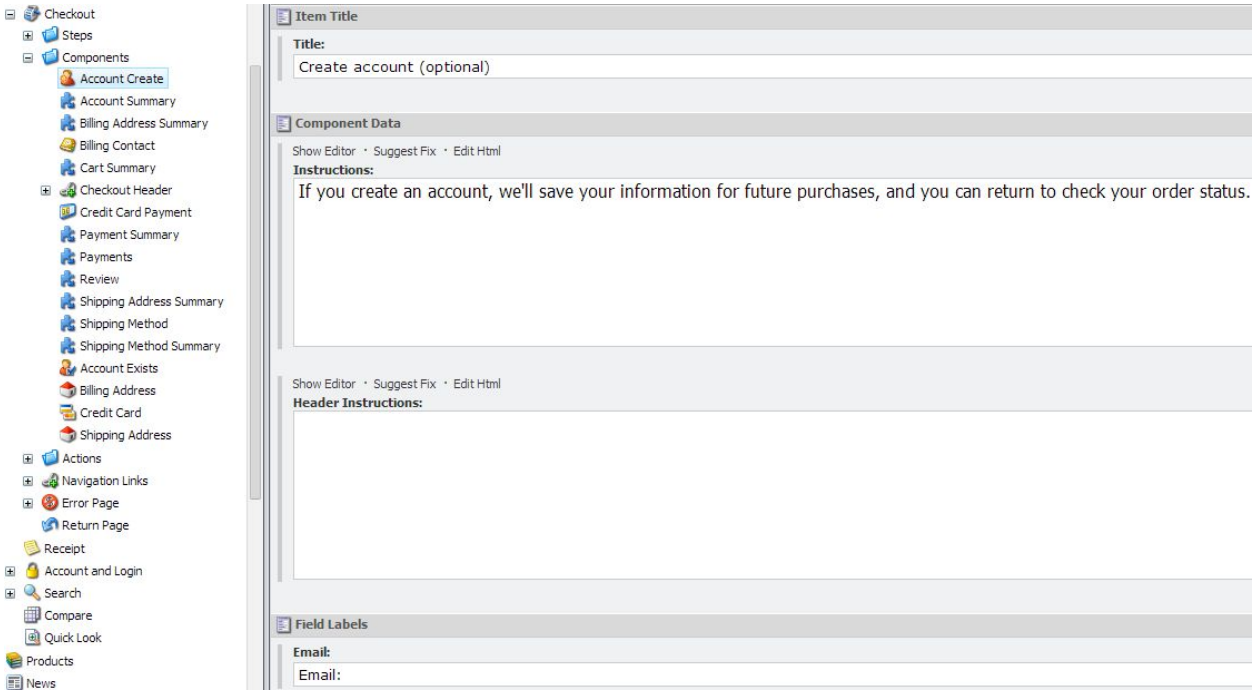
- *activated* - triggered when the Checkout Step has been activated
- *completed* - triggered when the Checkout Step has been completed

As an example, the shipping address component will want to validate all form fields are entered and also run address validation on the *validate* event. On the *process* event, it will call the appropriate service method to update the address on the server.

Datasource Item

All components use the *ActiveCommerce/Purchasing/CheckoutComponents/Checkout Component* template. This includes common fields like Title, Instructions, and Header Instructions. Often times, a component requires additional fields specific to the component type, in which case a derived template is used.

An example corresponding datasource item in Sitecore:



Actions

Actions are added to the *Checkout Step Actions* placeholder. These are essentially buttons which progress the customer through each step, triggering necessary events along the way.

Included actions are:

- *Next* - Continue to next step, triggering *validate* and then *process* events
- *Skip* - Skip the current step (no *validate* or *process* events triggered)
- *Place Order* - Place the order

Creating a Checkout Component

1. Create the View

- Decide if you need a new template for your component.
    - If you'll have additional text displayed on your component (field labels, etc), and you want those to be personalizable (in the Page Editor), create a template for your component.
    - On your new template, select *ActiveCommerce/Purchasing/CheckoutComponents/Checkout Component* as a base template.
- Create your user control (ascx file) and add it to the base directory of your skin (e.g. skins/sherpa/Checkout-Foo.ascx).
    - In the code-behind, change to inherit from ActiveCommerce.Web.skins.ActiveGlassUserControl<ActiveCommerce.Content.

Checkout.CheckoutComponents.CheckoutComponent> (or, a mapped class for your new template)
    - You can use the following code as a starting point (for a summary component, you can delete the HeaderInstructions and Instructions parts):

```
<div ng-controller="FooCtrl" class="foo checkout-component">

        <% if (!string.IsNullOrEmpty(Model.Title) ||
Sitecore.Context.PageMode.IsPageEditorEditing) { %>
        <h3><%=Editable(x => x.Title) %></h3>
        <% } %>

        <% if (!string.IsNullOrEmpty(Model.HeaderInstructions) ||
Sitecore.Context.PageMode.IsPageEditorEditing) { %>
        <div class="header-instructions"><%=Editable(x => x.HeaderInstructions)
%></div>
        <% } %>

        <% if (!string.IsNullOrEmpty(Model.Instructions) ||
Sitecore.Context.PageMode.IsPageEditorEditing) { %>
        <div class="instructions">
        <%=Editable(x => x.Instructions) %>
        </div>
        <% } %>

        <!-- COMPONENT MARKUP HERE -->

</div>
```

- Create the corresponding sublayout in Sitecore.
    - For **Datasource Location**, enter "./Components".
    - Select the **Datasource Tempate**. This will be either *ActiveCommerce/Purchasing/CheckoutComponents/Checkout Component* template, or your derived template
    - Enter the **Ascx file** location. E.g. /skins/~SKIN~/Checkout-Foo.ascx. Note the use of the skin token here (not required, but allows the file to be overridden in sub-skins)
    - For **Parameters**, enter "contentFolderTemplate={B7F10490-1B99-4BAA-872F-FE6FAB46A844}".
    - Add this new sublayout as an Allowed Control for the desired placeholder settings (i.e. ActiveCommerce/Checkout Step Panel or ActiveCommerce/Checkout Step Summary)

2. Create the Controller

- Create your AngularJs js file and add it to the checkout scripts directory of your skin (e.g.

skins/sherpa/scripts/checkout/<components/>foo.js).

- You can use the following code as a starting point (delete non-required event listeners):

```
angular.module('checkout.components')

	.controller('FooCtrl', ['$scope', 'checkout', function ($scope, checkout) {

	// INITIALIZE $SCOPE, ETC HERE

	// HANDLE APPROPRIATE EVENTS HERE
	$scope.$on("activate", function (e, args) {
		// Deferred event, so args.defer and args.reject methods available
	});

	$scope.$on("activated", function (e, args) {
		// Non-deferred event
	});

	$scope.$on("validate", function (e, args) {
		// Deferred event, so args.defer and args.reject methods available
	});

	$scope.$on("process", function (e, args) {
		// Deferred event, so args.defer and args.reject methods available
	});

	$scope.$on("completed", function (e, args) {
		// Non-deferred event
	});
	}]);
```

3. Add or Extend Services (Optional)

- You can either add new services, or extend existing services if necessary.
- If adding a new service, simply add it to the checkout scripts directory of your skin (e.g. skins/sherpa/scripts/checkout/<services/>myservice.js). This will automatically be included on the checkout page and will be available for injection into your controllers.
- Extending an existing service is built-in to AngularJs, using the $provide service. For example, if we want to extend the checkout service:

```
angular.module('checkout.services')

	.config(["$provide", function ($provide) {
	    $provide.decorator('checkout', ['$delegate', '$http', function ($delegate,
$http) {
		$delegate.updateShippingAddress = function (address) {
			return $http.post('/checkout/UpdateShippingAddress',
```

```
address).then(function (response) {
                        console.log("update has been decorated!");
                        });
                };
             $delegate.updateFoo = function(foo) {
                   console.log("foo has been updated!");
             };
                return $delegate;
          }]);
    }]);
```

- In this case, we're overriding the base `updateShippingAddress` method, and we're also adding a new `updateFoo` method.

Overriding Existing Checkout Components

Views

- To override a base view, simply use the skinning system. It is no different than overriding other Active Commerce sublayouts.
- See *Active Commerce Skinning Guide - Layouts & Sublayouts* section for more details.

Controllers

- AngularJs allows controller inheritance using the $controller service.
- $scope functions and properties can be overridden directly, and event listeners ($on) can be added. However, note that any base ones will still run.
- If necessary to remove these, you could include and use an "$off" implementation - an example of which can be found here.
- For example, if we want to extend the base ShippingAddressCtrl:

```
angular.module('checkout.components')

      .controller('MyShippingAddressCtrl', ['$scope', '$controller', function($scope,
$controller) {

      $controller('ShippingAddressCtrl', { $scope: $scope }); // Inherit from base
controller, injecting $scope

      $scope.countryChange = function() {
            console.log("different country change implementation here");
      };

      $scope.$on('process', function (e, args) {
            console.log("more processing to do");
      });
```

```
}]);
```

- In this case, we're overriding the base `countryChange` method and adding an additional `process` event handler.

Services

- See previous section "Add or Extend Services" for details on how to override an existing AngularJs service.

Example - Adding a Gift Message

In this example, we're going to add a gift message component. A text area will be added to the Shipping Info step where the customer can add an optional gift message. This will be saved along with our order information.

The complete code source is part of the Active Commerce Training solution on GitHub.

1. Extend the Checkout State Object

Active Commerce uses a session object which keeps track of the state of the checkout. We'll use this to temporarily store our gift message text.

- IGiftMessage - Adds our "GiftMessage" property
- CheckOut - Extend the base CheckOut object, adding our IGiftMessage interface
- Register in Unity - Use our version of ICheckOut

2. Extend the Checkout ViewModel

The CheckoutViewModel represents the checkout "state" that is passed via JSON to our front-end checkout components.

- CheckoutViewModel - Extend the base CheckoutViewModel, adding a GiftMessage model
- CheckoutViewModelFactory - Grabs our extended CheckOut object and passes on any stored GiftMessage to the extended CheckoutViewModel
- Register in Unity - Use our versions of CheckoutViewModel and CheckoutViewModelFactory

3. Extend the Checkout Controller

We need to add a new endpoint for our front-end service to call.

- CheckoutController - Extend the base CheckoutController, adding an "UpdateGiftMessage" action
- RegisterRoutesInitializeProcessor - Register the MVC "checkout" route using our new

Checkout controller
- Config patch - Patch in our RegisterRoutesInitializeProcessor

4. Extend the Order

We'd like the entered gift message to be added to the order details, so we have to extend the base Order domain model.

- Order - Extend the base Order class, adding our "GiftMessage" public virtual property
- OrderMap - Create an OrderMap class which inherits FluentNHibernate.Mapping.SubclassMap with your Order class as the generic argument. In the constructor for your OrderMap, use Fluent NHibernate mapping syntax to map your new properties
- Config patch - Create a configuration patch which registers your assembly in the acConfigurationBuilder pipeline, so that your NHibernate mappings are loaded on startup
- Ensure your database schema has the needed table and fields. In a development environment, you can use /sitecore/admin/DatabaseUtility.aspx to update or drop/create the schema. For production, either generate the needed update script or use a SQL comparison tool
- OrderFactory - Create an OrderFactory class which inherits ActiveCommerce.Orders.OrderFactory. This class is used by Active Commerce to construct new orders. Override the CreateOrder method and return your new order type
- Register in Unity - In a new or existing ITypeRegistration class, register your OrderFactory with the container as the implementation of the IOrderFactory service

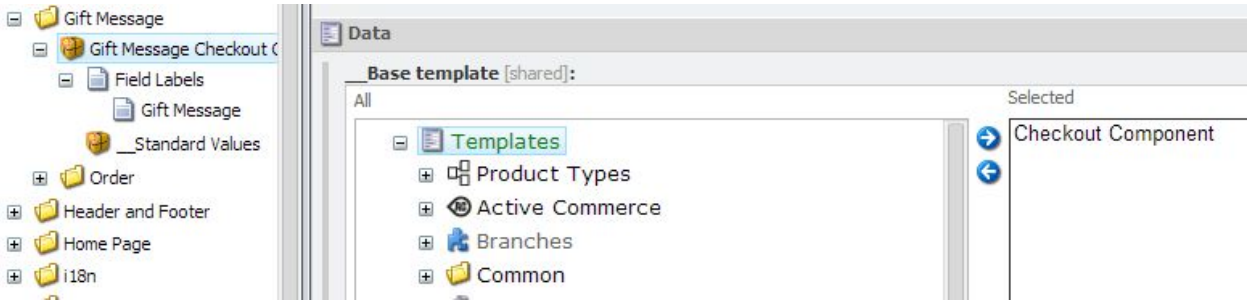
5. Add an OrderPipelineProcessor

- SaveGiftMessage - Takes the GiftMessage from our extended CheckOut object on the args and sets the GiftMessage on our extended Order
- Config patch - Add our processor to the *acStartOrderProcessing* pipeline

6. Create the Checkout Components

Now we'll create the actual checkout components that will be placed on the checkout page. Refer to the section *Creating a Checkout Component* above for more information on this process.

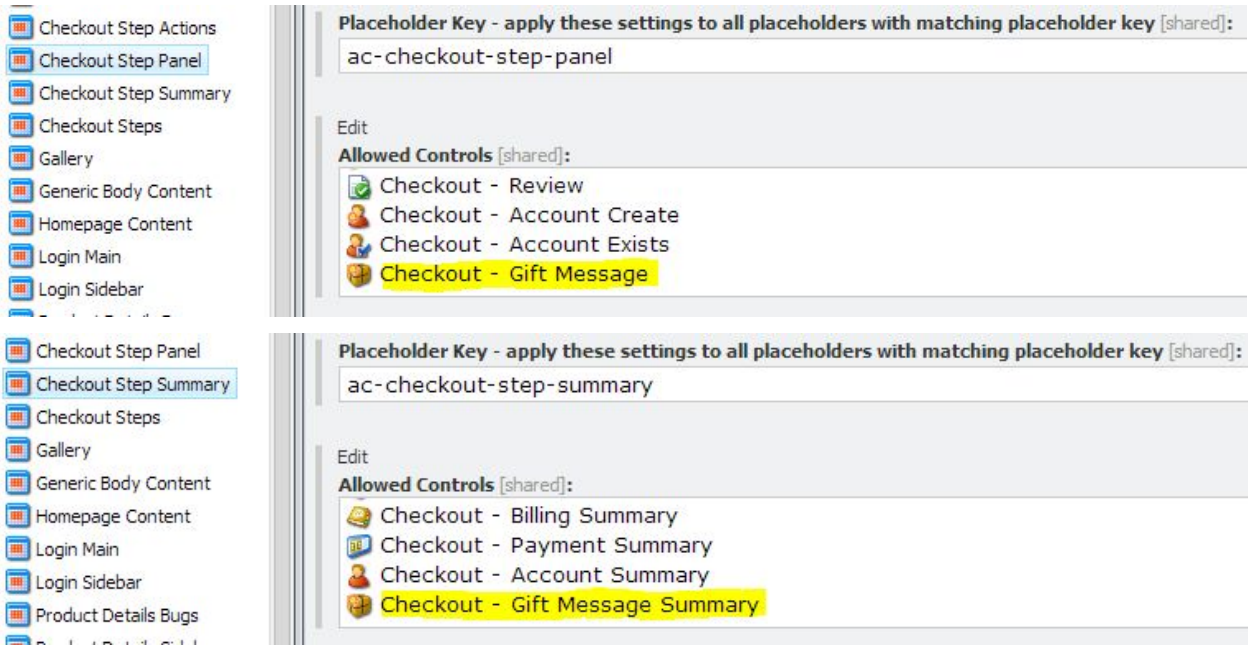
- Template - Inherit the base Checkout Component template (*/ActiveCommerce/Purchasing/CheckoutComponents/Checkout Component*), adding a "Gift Message" label field



- GiftMessageCheckoutComponent - Create mapped concrete class for our new template (using Glass Mapper)
- View + Code Behind - Contains gift message form to present to customer
- Summary View + Code Behind - Contains summary view of gift message component
- Sublayout, Summary Sublayout - Corresponding sublayouts in Sitecore
- Controller, Summary Controller - Corresponding AngularJs controllers for our components
- Service - Extend the base AngularJs “checkout” service, adding an “updateGiftMessage” method.
- Config patch - Tell Glass Mapper where to find our Glass mapped class

7. Add Components to the Checkout Page

- First, we need to add sublayouts to the appropriate placeholder settings. In this case, we want to be able to add our Gift Message component to the *Checkout Step Panel* placeholder, and our Gift Message Summary component to the *Checkout Step Summary* placeholder.



- Finally, add our gift message components to checkout page via Page Editor (w/ new datasource items)