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 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 online 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).
- Here's an example of our CyberSource plugin:

```
public class PaymentProvider : IntegratedPaymentProviderBase, ICreditable,
IReservable
{
	protected const string SETTING_AUTHORIZE_ONLY = "authorizeOnly";
	protected const string DECISION_ACCEPT = "ACCEPT";
```

```
        protected const string DECISION_ERROR = "ERROR";
        protected const string DECISION_REJECT = "REJECT";

        public override void Invoke(PaymentSystem paymentSystem, PaymentArgs
paymentArgs)
        {
        var cart = paymentArgs.ShoppingCart as ShoppingCart;
        if (cart == null)
        {
              throw new Exception("Cart must be of type
ActiveCommerce.Carts.ShoppingCart");
        }
        var settings = GetSettings(paymentSystem);

        var request = new RequestMessage
        {
             merchantID = paymentSystem.Username,
             merchantReferenceCode = cart.OrderNumber,
             billTo = new BillTo
              {
                 firstName = cart.CustomerInfo.BillingAddress.Name,
                 lastName = cart.CustomerInfo.BillingAddress.Name2,
                 street1 = cart.CustomerInfo.BillingAddress.Address,
                 city = cart.CustomerInfo.BillingAddress.City,
                 state = cart.CustomerInfo.BillingAddress.State,
                 postalCode = cart.CustomerInfo.BillingAddress.Zip,
                 country = cart.CustomerInfo.BillingAddress.Country.Code,
                 email = cart.CustomerInfo.Email,
                 phoneNumber = cart.CustomerInfo.BillingAddress.GetPhoneNumber()
             },
             card = new Card
              {
              accountNumber = cart.CreditCardInfo.CardNumber.ToString(),
                 expirationMonth = cart.CreditCardInfo.ExpirationDate.ToString("MM"),
                 expirationYear = cart.CreditCardInfo.ExpirationDate.ToString("yyyy"),
                 cardType = GetCardTypeId(cart.CreditCardInfo.CardType)
             },
             purchaseTotals = new PurchaseTotals
              {
                 currency = cart.Currency.Code,
                 grandTotalAmount = cart.Totals.TotalPriceIncVat.ToString("0.00") //
Uses midpoint away from zero rounding
              }
        };

        // For CyberSource (Them: "To help us troubleshoot any problems that you may
encounter, please include the following information about your application.")
          request.clientLibrary = ".NET WCF";
```

```
        request.clientLibraryVersion = Environment.Version.ToString();

request.clientEnvironment = Environment.OSVersion.Platform +
Environment.OSVersion.Version.ToString();

       // Turn on Authorize
        request.ccAuthService = new CCAuthService {run = "true"};
       if (settings.IncludeCapture)
       {
              // Turn on Capture
             request.ccCaptureService = new CCCaptureService {run = "true"};
       }

       var reply = MakeRequest(request, paymentSystem, false /* CheckPaymentStatus
will log error */);
       if (reply.decision != DECISION_ACCEPT) return;

       var transactionData = Context.Entity.Resolve<ITransactionData>();
       if (settings.IncludeCapture)
       {
             PaymentStatus = PaymentStatus.Captured;
       }
       else
       {
              var reservationTicket = new ReservationTicket
                                          {
                                                InvoiceNumber = cart.OrderNumber,

Amount = Math.Round(cart.Totals.TotalPriceIncVat, 2, MidpointRounding.AwayFromZero),

AuthorizationCode = reply.ccAuthReply.authorizationCode,
                                                TransactionNumber = reply.requestID
                                          };

transactionData.SavePersistentValue(cart.OrderNumber, "ReservationTicket",
reservationTicket);
             PaymentStatus = PaymentStatus.Reserved;
       }

        transactionData.SaveCallBackValues(cart.OrderNumber,
                                          PaymentStatus.ToString(),
                                          reply.requestID,

cart.Totals.TotalPriceIncVat.ToString("0.00"),
                                          cart.Currency.Code,
                                          reply.decision, //SEFE bug. pass in here,
get via PaymentStatus
```

```
                                                reply.reasonCode,
                                                reply.reasonCode,
                                                cart.CreditCardInfo.CardType);

transactionData.SavePersistentValue(cart.OrderNumber,
TransactionConstants.AuthorizationNumber, reply.ccAuthReply.authorizationCode);

      }

    #region IReservable Members

      public void Capture(PaymentSystem paymentSystem, PaymentArgs paymentArgs,
ReservationTicket reservationTicket, decimal amount)
      {
      var request = new RequestMessage
      {
            merchantID = paymentSystem.Username,
            merchantReferenceCode = reservationTicket.InvoiceNumber,
            ccCaptureService = new CCCaptureService
             {
                run = "true",
                authRequestID = reservationTicket.TransactionNumber
            },
            purchaseTotals = new PurchaseTotals
             {
                currency = paymentArgs.ShoppingCart.Currency.Code,
                grandTotalAmount = reservationTicket.Amount.ToString("0.00") // Uses
away from zero rounding
             }
      };

      var reply = MakeRequest(request, paymentSystem);
      if (reply.decision != DECISION_ACCEPT) return;

      var transactionData = Context.Entity.Resolve<ITransactionData>();
        transactionData.SavePersistentValue(reservationTicket.InvoiceNumber,
PaymentConstants.CaptureSuccess);
      }

      public void CancelReservation(PaymentSystem paymentSystem, PaymentArgs
paymentArgs, ReservationTicket reservationTicket)
      {
      var request = new RequestMessage
      {
            merchantID = paymentSystem.Username,
            merchantReferenceCode = reservationTicket.InvoiceNumber,
            voidService = new VoidService
```

```
            {
               run = "true",
               voidRequestID = reservationTicket.TransactionNumber
            }
      };

      var reply = MakeRequest(request, paymentSystem);
      if (reply.decision != DECISION_ACCEPT) return;

      var transactionData = Context.Entity.Resolve<ITransactionData>();
       transactionData.SavePersistentValue(reservationTicket.InvoiceNumber,
PaymentConstants.CancelSuccess);
      }

   #endregion

   #region ICreditable Members

      public void Credit(PaymentSystem paymentSystem, PaymentArgs paymentArgs,
ReservationTicket reservationTicket)
      {
      var request = new RequestMessage
      {
            merchantID = paymentSystem.Username,
            merchantReferenceCode = reservationTicket.InvoiceNumber,
            ccCreditService = new CCCreditService
             {
               run = "true",
               captureRequestID = reservationTicket.TransactionNumber
            },
            purchaseTotals = new PurchaseTotals
             {
               currency = paymentArgs.ShoppingCart.Currency.Code,
               grandTotalAmount = reservationTicket.Amount.ToString("0.00") // Uses
away from zero rounding
            }
      };

      var reply = MakeRequest(request, paymentSystem);
      if (reply.decision != DECISION_ACCEPT) return;

      var transactionData = Context.Entity.Resolve<ITransactionData>();
       transactionData.SavePersistentValue(reservationTicket.InvoiceNumber,
PaymentConstants.CreditSuccess);
      }

   #endregion
```

```
        protected ReplyMessage MakeRequest(RequestMessage request, PaymentSystem
paymentSystem, bool logError = true)
        {
        try
        {
                var proc = new TransactionProcessorClient();

               proc.ChannelFactory.Credentials.UserName.UserName = request.merchantID;

proc.ChannelFactory.Credentials.UserName.Password = paymentSystem.Password;

                var reply = proc.runTransaction(request);

                if (reply.decision != DECISION_ACCEPT)
                {
                   var message = new StringBuilder();

if (reply.ccAuthReply != null && !
String.IsNullOrWhiteSpace(reply.ccAuthReply.reasonCode))
                    {
                        message.AppendFormat("[Auth]:{0}", reply.ccAuthReply.reasonCode);
                    }

if (reply.ccCaptureReply != null && !
String.IsNullOrWhiteSpace(reply.ccCaptureReply.reasonCode))
                    {

message.AppendFormat("[Capture]:{0}", reply.ccCaptureReply.reasonCode);
                    }

if (reply.ccCreditReply != null && !
String.IsNullOrWhiteSpace(reply.ccCreditReply.reasonCode))
                    {

message.AppendFormat("[Credit]:{0}", reply.ccCreditReply.reasonCode);
                    }
                    var transactionData = Context.Entity.Resolve<ITransactionData>();
                    transactionData.SaveCallBackValues(request.merchantReferenceCode, //
OrderNumber
                                                       PaymentStatus.Failure.ToString(),
                                                       reply.requestID,
                                                       reply.purchaseTotals != null ?
reply.purchaseTotals.grandTotalAmount : null,
                                                       reply.purchaseTotals != null ?
reply.purchaseTotals.currency : null,
```

```
                                                    reply.decision, //SEFE bug. pass
in here, get via PaymentStatus
                                                    reply.reasonCode,
                                                    message.ToString(),
                                                    request.card != null ?
request.card.cardType : null);

                PaymentStatus = PaymentStatus.Failure;

                if (logError)
                {
                    Log.Error(string.Format(
                          "CyberSource transaction failure: {0}\nProvider Message =
{1}\nProvider Error Code = {2}\nTransaction Number = {3}",
                          reply.decision, message, reply.reasonCode,
reply.requestID), this);

                }
             }
             return reply;
       }
           catch (Exception e)
       {
             Log.Error("CyberSource transaction failure", e, this);
            PaymentStatus = PaymentStatus.Failure;
             return new ReplyMessage {decision = DECISION_ERROR};
       }
       }

       protected static string GetCardTypeId(string cardType)
       {
       switch (cardType.ToLowerInvariant())
       {
             case "visa":
                return "001";
             case "mastercard":
                return "002";
             case "american express":
                  return "003";
             case "discover":
                return "004";
             case "diners club":
                return "005";
             case "carte blanche":
                return "006";
             case "jcb":
                return "007";
```

```
            default:
               return null;
        }
        }

        protected Settings GetSettings(PaymentSystem paymentSystem)
        {
        var settings = new Settings();
        var settingsReader = new
Sitecore.Ecommerce.Payments.PaymentSettingsReader(paymentSystem);

        //a little backwards, but we want to configure for authorize only, not for
include capture. capture unless auth only explicitly set to "true"
        var authorizeSetting = settingsReader.GetSetting(SETTING_AUTHORIZE_ONLY);

settings.IncludeCapture = authorizeSetting == null || (!
Boolean.TrueString.ToLower().Equals(authorizeSetting.ToLower()));

        return settings;
        }

        protected struct Settings
        {
        public bool IncludeCapture;
        }
}
```

- You could also check out the SES ones: https://sdn.sitecore.net/Products/SEFE/SES12/Payment%20Providers/120_111101.aspx

2. Register with Unity

- Make sure to register it as a named instance (e.g. "CyberSource").
- For example, here is the registration for CyberSource plugin:

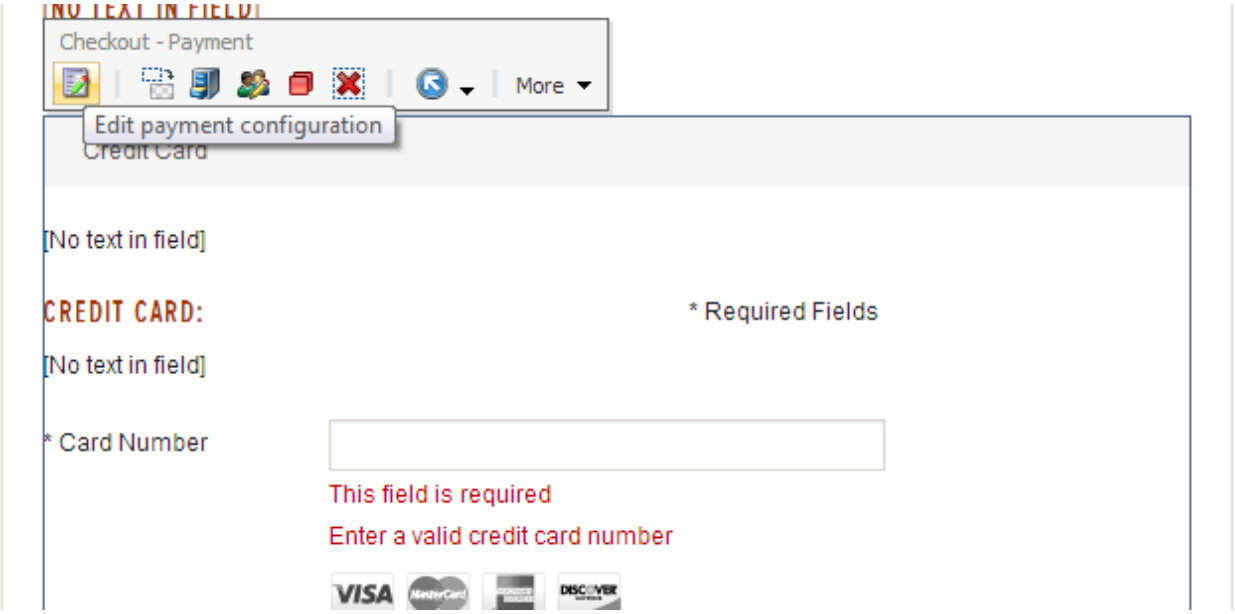
```
container.RegisterType(typeof(Sitecore.Ecommerce.DomainModel.Payments.PaymentProvider
),
                                typeof(PaymentProvider), "CyberSource");
```

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. "CyberSource").
- 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 3rd party service).

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 full details. You’ll follow the same process.

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

2. Register with Unity

- Make sure to register it as a named instance (e.g. "FedEx").
- For example, here is the registration for FedEx plugin:

```
container.RegisterType(typeof(ActiveCommerce.Shipping.IShippingService),

typeof(ActiveCommerce.FedEx.ShippingService), "FedEx", new InjectionMember[] {
new InjectionProperty("Configuration"),
new InjectionProperty("Destination"),
new InjectionProperty("ShippingWeight")
});
```

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.

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

- Here's an example of our VatTaxCalculator:

```
public class VatTaxCalculator : TaxCalculatorBase<TaxConfiguration>
{
	public const string JURISDICTION_TYPE = "VAT";

	public override TaxTotals GetTaxes(IEnumerable<TaxInquiry> order)
	{
	var rounder = Sitecore.Ecommerce.Context.Entity.Resolve<ICurrencyRounder>();
	order = order.ToList();
	return new TaxTotals
	{
		ProductTax = order.Where(x => !x.IsShipping && !x.IsHandling)
				.Select(x => new { TaxInquiry = x, Product =
GetProduct(x.ProductCode) })
				.Select(x => new { TaxInquiry = x.TaxInquiry,
Product = x.Product, Rate = GetRate(x.Product.TaxType) })
				.Select(x => new TaxLine
						{
						    ProductCode = x.Product.Code,
						    Type = x.Product.TaxType !=
null ? x.Product.TaxType.Code : string.Empty,
						    TaxedAmount = x.TaxInquiry.Total,

Jurisdictions = new TaxJurisdiction
                                                                {
                                                                    Name =
Config.Name,
                                                                    Type =
TaxJurisdiction.Types.VAT,
                                                                    Rate =
x.Rate,
                                                                    Tax =
rounder.Round(x.Rate * x.TaxInquiry.Total, Currency)
```

```
 }.ToEnumerable()
                                                 }),
            ShippingTax = order.Where(x => x.IsShipping)
                                .Select(x => new { TaxInquiry = x, Rate =
GetRate(Config.ShippingVatType)})
                                .Where(x => x.Rate > 0)
                              .Select(x => new TaxLine
                                                 {

Type = Config.ShippingVatType.Code,
                                                      TaxedAmount = x.TaxInquiry.Total,
                                                       Jurisdictions = new
TaxJurisdiction
                                                                                {
                                                                                    Name =
Config.Name,
                                                                                    Type =
JURISDICTION_TYPE,
                                                                                    Rate =
x.Rate,
                                                                                    Tax =
rounder.Round(x.Rate * x.TaxInquiry.Total, Currency)

 }.ToEnumerable()
                                                 }).FirstOrDefault(),
            HandlingTax = order.Where(x => x.IsHandling)
                                .Select(x => new { TaxInquiry = x, Rate =
GetRate(Config.HandlingVatType)})
                                .Where(x => x.Rate > 0)
                                .Select(x => new TaxLine
                                                 {

Type = Config.ShippingVatType.Code,
                                                      TaxedAmount = x.TaxInquiry.Total,
                                                       Jurisdictions = new
TaxJurisdiction
                                                                                {
                                                                                    Name =
Config.Name,
                                                                                   Type =
JURISDICTION_TYPE,
                                                                                    Rate =
x.Rate,
                                                                                    Tax =
rounder.Round(x.Rate * x.TaxInquiry.Total, Currency)

 }.ToEnumerable()
```

```
                                                    }).FirstOrDefault()
        };
        }

        public override bool WillHandle()
    {
        return MatchesLocation(Shipping);
        }


        protected virtual Product GetProduct(string productCode)
        {
        var repository =
Sitecore.Ecommerce.Context.Entity.Resolve<IProductRepository>();
        var product = repository.Get<Product>(productCode);
        return product;
        }

        protected virtual decimal GetRate(TaxType taxType)
        {
        if (taxType == null || !taxType.Values.Any())
        {
                return 0m;
        }
        var taxValue = taxType.Values.FirstOrDefault(x => x.TaxConfiguration != null
&& x.TaxConfiguration.Code == Config.Code);
        if (taxValue == null)
        {
                return 0m;
        }
        return taxValue.Rate;
        }
}
```

2. Register with Unity

- Make sure to register it as a named instance (e.g. "MyTaxCalc").

```
container.RegisterType(typeof(ITaxCalculator), typeof(Foo.Bar.MyTaxCalculator),
"MyTaxCalc")
```

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).
- Here's an example of our FastTax plugin:

```
public class SalesTaxRateProvider : ISalesTaxRateProvider
{
	protected static Cache _cache;
	protected static object _cacheLock = new object();

	protected const string TAX_TYPE = "sales";
	protected const string UNITED_STATES = "US";
	protected const string CANADA = "CA";
	protected const string LICENSE_KEY_SETTING
= "ActiveCommerce.FastTax.LicenseKey";
	protected const string FAST_TAX_CACHE_NAME = "ActiveCommerce.FastTax.Rates";
	protected const string FAST_TAX_CACHE_SIZE_SETTING
= "ActiveCommerce.FastTax.Cache.Size";

	protected string _licenseKey;

	protected Cache Cache
	{
	get
	{
		lock (_cacheLock)
		{
		   if (_cache == null)
		   {
```

```
_cache = Sitecore.Caching.Cache.GetNamedInstance(FAST_TAX_CACHE_NAME,

      Sitecore.StringUtil.ParseSizeString(Sitecore.Configuration.Settings.GetSetting
(FAST_TAX_CACHE_SIZE_SETTING)));
                }
              }
             return _cache;
      }
      }

      public SalesTaxRateProvider()
      {

_licenseKey = Sitecore.Configuration.Settings.GetSetting(LICENSE_KEY_SETTING);
      }

    #region ISalesTaxRateProvider Members

      public IEnumerable<TaxRate> GetRates(AddressInfo address)
      {
      if (address.Country == null)
      {
            throw new Exception("No country on the provided address");
      }

      string cacheKey = GetCacheKey(address);
      var cachedRates = Cache.GetEntry(cacheKey, true);
      if (cachedRates != null)
      {
            return cachedRates.Data as IEnumerable<TaxRate>;
      }

      IEnumerable<TaxRate> rates = null;
      if (address.Country.Code == UNITED_STATES)
      {
           rates = GetUnitedStatesTaxRates(address);
      }
      else if (address.Country.Code == CANADA)
      {
           rates = GetCanadianTaxRates(address);
      }
      else
      {
            throw new Exception("This rate provider only supports the United States
and Canada");
      }
```

```
        if (rates != null)
        {
              Cache.Add(cacheKey, rates, GetDataSize(rates), DateTime.UtcNow.AddDays(1)
);
        }

        return rates;
        }

    #endregion

        public IEnumerable<TaxRate> GetUnitedStatesTaxRates(AddressInfo address)
        {
        var service = new DOTSFastTaxSoapClient();

        if (address.Zip.IsNullOrEmpty())
        {
               throw new Exception("U.S. sales tax lookup requires zip code");
        }

        TaxInfo taxInfo;
        if (address.Address.IsNullOrEmpty() || address.City.IsNullOrEmpty())
        {
              taxInfo = GetTaxInfoForZip(address.Zip);
        }
        else
        {
               try
               {
                  taxInfo = GetTaxInfoForAddress(address);
               }
               catch (FastTaxException exception)
               {
                  Sitecore.Diagnostics.Log.Warn("Exception getting sales tax for
address {0}, will attempt by zip alone".FormatWith(GetCacheKey(address)), exception,
this);
                  taxInfo = GetTaxInfoForZip(address.Zip);
               }
        }
        return GetRatesForTaxInfo(taxInfo);
        }

        public IEnumerable<TaxRate> GetCanadianTaxRates(AddressInfo address)
        {
        var service = new DOTSFastTaxSoapClient();
```

```
        if (address.State.IsNullOrEmpty())
        {
                throw new Exception("Canadian sales tax lookup requires province.");
        }

        var taxResponse = service.GetCanadianTaxInfoByProvince(address.State,
_licenseKey);
        if (taxResponse.Error.IsError())
        {
                throw new FastTaxException(taxResponse.Error);
        }

        var rates = new List<TaxRate>();
        if (taxResponse.HarmonizedSalesTax > 0)
        {
              rates.Add(new TaxRate
                              {
                                      Name = taxResponse.ProvinceAbbreviation,
                                      Type = TaxJurisdiction.Types.CANADA_HST,
                                      Rate = taxResponse.HarmonizedSalesTax
                              });
        }
        else if (taxResponse.ProvinceSalesTax > 0)
        {
              rates.Add(new TaxRate
                              {
                                      Name = CANADA,
                                       Type = TaxJurisdiction.Types.CANADA_GST,
                                      Rate = taxResponse.GoodsSalesTax
                              });
              rates.Add(new TaxRate
                              {
                                      Name = taxResponse.ProvinceAbbreviation,
                                      Type = TaxJurisdiction.Types.CANADA_PST,
                                      Rate = taxResponse.ProvinceSalesTax,
                                      Compounds =
Boolean.TrueString.Equals(taxResponse.ApplyGSTFirst,
StringComparison.InvariantCultureIgnoreCase)
                              });
        }
        return rates.AsEnumerable();
        }

        protected TaxInfo GetTaxInfoForZip(string zipCode)
        {
        var service = new DOTSFastTaxSoapClient();
        var taxResponse = service.GetTaxInfoByZip_V2(zipCode, TAX_TYPE, _licenseKey);
        if (taxResponse.Error.IsError())
```

```
        {
                throw new FastTaxException(taxResponse.Error);
        }
        if (taxResponse.TaxInfo == null || taxResponse.TaxInfo.Length == 0)
        {
                throw new FastTaxException(string.Format("Fast Tax returned empty tax
info for {0}", zipCode));
        }
        return taxResponse.TaxInfo[0];
        }

        protected TaxInfo GetTaxInfoForAddress(AddressInfo address)
        {
        var service = new DOTSFastTaxSoapClient();
        var taxInfo = service.GetTaxInfoByAddress(address.Address, address.City,
address.State, address.Zip, TAX_TYPE, _licenseKey);
        if (taxInfo.Error.IsError())
        {
                throw new FastTaxException(taxInfo.Error);
        }
        return taxInfo;
        }

        protected IEnumerable<TaxRate> GetRatesForTaxInfo(TaxInfo taxInfo)
        {
        var rates = new List<TaxRate>();
        if (taxInfo.StateRate > 0)
        {
              rates.Add(new TaxRate
                              {
                                      Name = taxInfo.StateAbbreviation,
                                      Type = TaxJurisdiction.Types.STATE,
                                      Rate = taxInfo.StateRate
                              });
        }
        if (taxInfo.CountyRate > 0)
        {
              rates.Add(new TaxRate
                              {
                                      Name = taxInfo.County,
                                      Type = TaxJurisdiction.Types.COUNTY,
                                      Rate = taxInfo.CountyRate
                              });
        }
        if (taxInfo.CountyDistrictRate > 0)
        {
              rates.Add(new TaxRate
                              {
```

```
                                Name = taxInfo.County,
                                  Type = TaxJurisdiction.Types.COUNTY_DISTRICT,
                                  Rate = taxInfo.CountyDistrictRate
                            });
      }
      if (taxInfo.CityRate > 0)
      {
            rates.Add(new TaxRate
                            {
                                  Name = taxInfo.City,
                                  Type = TaxJurisdiction.Types.CITY,
                                  Rate = taxInfo.CityRate
                            });
      }
      if (taxInfo.CityDistrictRate > 0)
      {
            rates.Add(new TaxRate
                            {
                                  Name = taxInfo.City,
                                  Type = TaxJurisdiction.Types.CITY_DISTRICT,
                                  Rate = taxInfo.CityDistrictRate
                            });
      }
      return rates.AsEnumerable();
      }

      public string GetCacheKey(AddressInfo address)
      {
      return address.Country.Code + "~" + address.Address + "~" + address.City
+ "~" + address.State + "~" + address.Zip;
      }

      public long GetDataSize(IEnumerable<TaxRate> rates)
      {
      rates = rates.ToList();
      //decimal is 128-bit / 16 bytes
      return rates.Sum(x => x.Name.Length) + rates.Sum(x => x.Type.Length) +
(rates.Count()*16);
      }

}
```

2. Register with Unity

- Make sure to register it as a named instance (e.g. "FastTax").

```
container.RegisterType(typeof(ISalesTaxRateProvider),
```

```
        typeof(ActiveCommerce.FastTax.SalesTaxRateProvider), "FastTax");
```

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. "FastTax").

Adding a Custom Address Validator

1. Create the Address Validator class

- Inherit from ActiveCommerce.Validation.IAddressValidator.
- Implement the required method(s).
- Here's an example of our default SimpleAddressValidator:

```
public class SimpleAddressValidator : IAddressValidator
{
        public const string US_ZIP_REGEX = @"^\d{5}(-\d{4})?$";
        public const string CANADA_ZIP_REGEX = @"^[ABCEGHJKLMNPRSTVXY]{1}\d{1}[A-Z]{1}
*\d{1}[A-Z]{1}\d{1}$";

        public Sitecore.Ecommerce.DomainModel.Addresses.AddressInfo
Validate(Sitecore.Ecommerce.DomainModel.Addresses.AddressInfo address)
        {
        if (String.IsNullOrWhiteSpace(address.Address) ||
              String.IsNullOrWhiteSpace(address.City) ||
             address.Country == null)
        {
              throw new AddressValidationException("Street, City, and Country are
required.");
        }

        // Perform some basic housekeeping
          address.Address = address.Address.Trim();
          address.Address2 = address.Address2 != null ? address.Address2.Trim() : null;
          address.City = address.City.Trim();
          address.State = address.State.Trim();
          address.Zip = address.Zip.Trim();

        // US validation
        if (address.Country.Code.Equals("US",
StringComparison.InvariantCultureIgnoreCase))
          {
              if (!(new Regex(US_ZIP_REGEX)).IsMatch(address.Zip))
              {
                 throw new AddressValidationException("Not a valid US zip code");
```

```
            }

            if (String.IsNullOrWhiteSpace(address.State))
            {
                 throw new AddressValidationException("State is required");
            }
      }

      // Canada validation
      if (address.Country.Code.Equals("CA",
StringComparison.InvariantCultureIgnoreCase))
      {
            if (!(new Regex(CANADA_ZIP_REGEX)).IsMatch(address.Zip))
            {
               throw new AddressValidationException("Not a valid Canadian zip
code");
            }

            if (String.IsNullOrWhiteSpace(address.State))
            {
               throw new AddressValidationException("Province is required");
            }
      }

      return address;
      }
}
```

2. Register with Unity

```
      container.RegisterType(typeof(ActiveCommerce.Validation.IAddressValidator),
typeof(Foo.Bar.AddressValidator));
```

Adding a Product Detail Tab

1. Create a user control

- Inherit from ActiveCommerce.Web.skins.ProductDetails_Base and ActiveCommerce.Web.skins.ITabControl
- Implement the required properties.
    - “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)
- Here is an example:

```
public partial class ProductTab_Info : ProductDetails_Base, ITabControl
{
	public string TitleKey
	{
	get { return "Product-Tab-Info"; }
	}

	public string Id
	{
	get { return "Info"; }
	}
}
```

- Use the following scaffolding for you .ascx code:

```
<% if (Sitecore.Context.PageMode.IsPageEditor) { %>
<ul class="ui-tabs-nav">
	<li class="ui-state-active"><a
href="#<%=Id %>"><%=Translator.Render((TitleKey)) %></a></li>
</ul>
<% } %>

<div id="<%=Id %>" class="ui-tabs-panel">
	<div class="body">
		<!--YOUR TAB CODE HERE-->
	</div>
</div>
```

- This ensures your new tab is usable in the Page Editor

2. Create a sublayout

- 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
- Use sorting to define where your tab appears (look for “Product Tab - “, these are the default Active Commerce tabs)

Adding an Order Pipeline Processor

1. Create a class

- Inherit from ActiveCommerce.OrderProcessing.IOrderPipelineProcessor
- Implement the required “Process” method

- Use OrderPipelineArgs argument as necessary
  - Cart 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 optionally AddMessage appropriately, and then call AbortPipeline()

2. Inject the processor into the orderProcessing Pipeline

- An example config file:

```
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
    <sitecore>
        <processors>
             <orderProcessing>
             <processor type="MySite.OrderProcessing.SomeAdditionalCheck,
MySite.Classes"

       patch:after="processor[@type='ActiveCommerce.OrderProcessing.CreateOrder,
ActiveCommerce.Kernel']"/>
             </orderProcessing>
        </processors>
    </sitecore>
</configuration>
```

Adding a Custom Order Status

1. Create the order status class

- Inherit from Sitecore.Ecommerce.Orders.Statuses.OrderStatusBase.
- Implement the required method(s).
- Here's an example:

```
public class Shipped : Sitecore.Ecommerce.Orders.Statuses.OrderStatusBase
{
       protected override void Process<T>(T order)
       {
             // Do nothing.
       }
}
```

2. Register with Unity

```
       container.RegisterType(typeof(Sitecore.Ecommerce.DomainModel.Orders.OrderStatu
s), typeof(Foo.Bar.Orders.Statuses.Shipped), "Shipped");
```

3. Add to Order Statuses

- In Sitecore, add a new Order Status to Webshop Business Settings/Order Status

- Set the "Code" field to the same name you registered with Unity (e.g. "Shipped"). Optionally, assign any next available statuses using the “Available List” field.

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

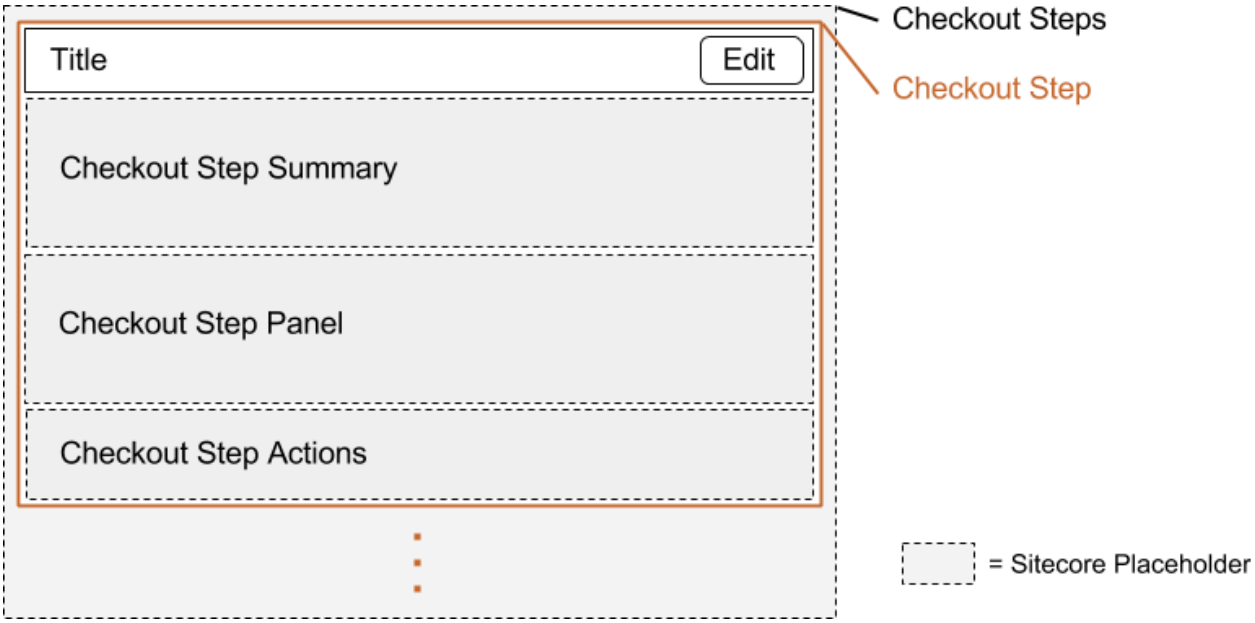
Architecture

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:

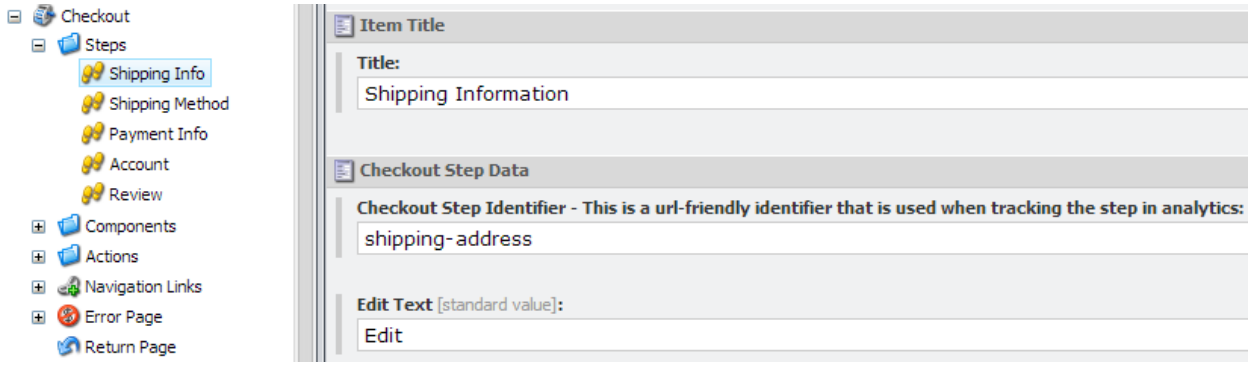
- The Edit button - The edit button will display once the step is *complete* and the step is also *editable*.
- The placeholders visibility - See *Placeholders*
- CSS - Classes (i.e. "active", "complete", "editable") are added to the checkout step container when in a given state

Datasource Item

Steps use the *ActiveCommerce/Purchasing/CheckoutComponents/Checkout Step* template.

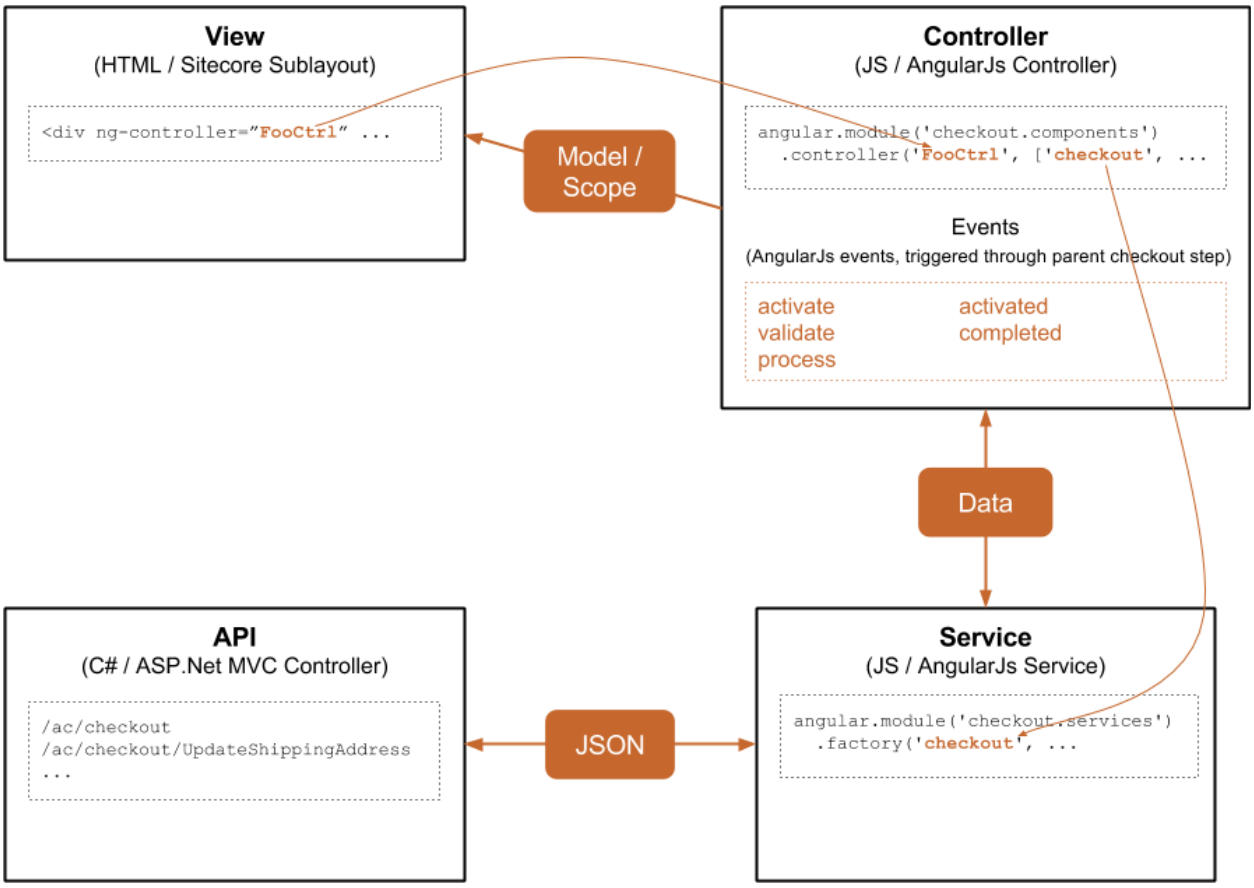
Note the “Checkout Step Identifier” field. This is used to uniquely identify the step in analytics, and also in javascript.

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:

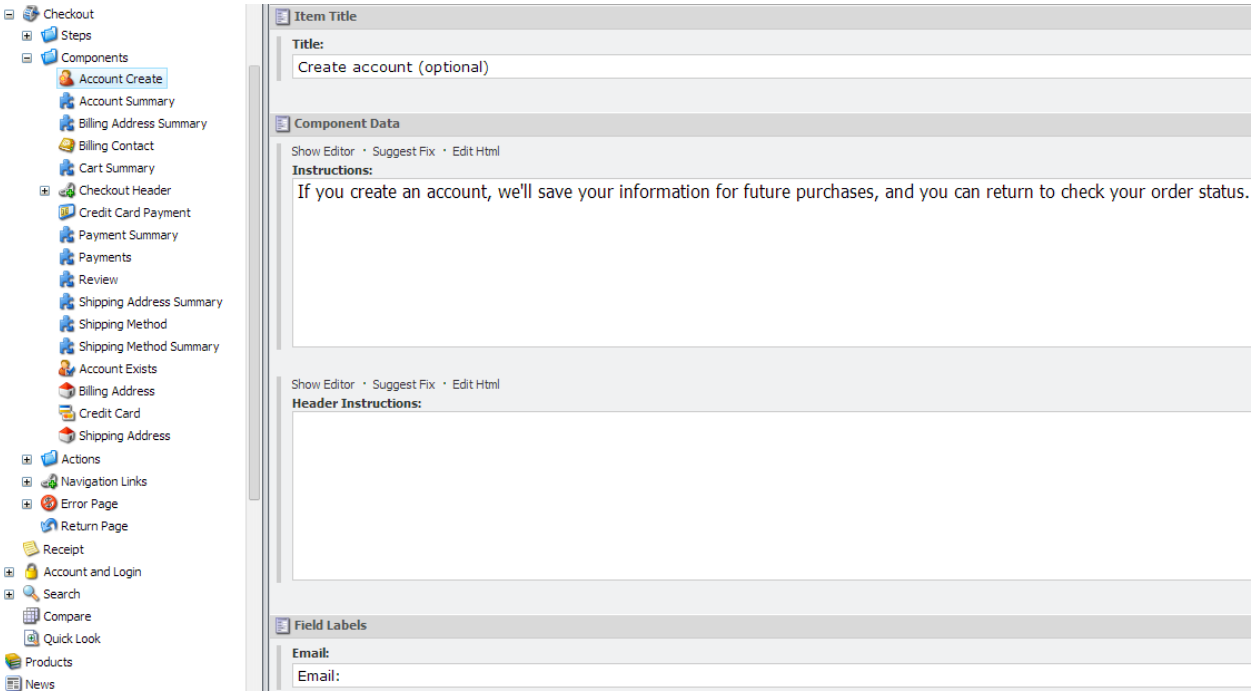
- *activated* - triggered when the Checkout Step has been activated
- *completed* - triggered when the Checkout Step has been completed

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

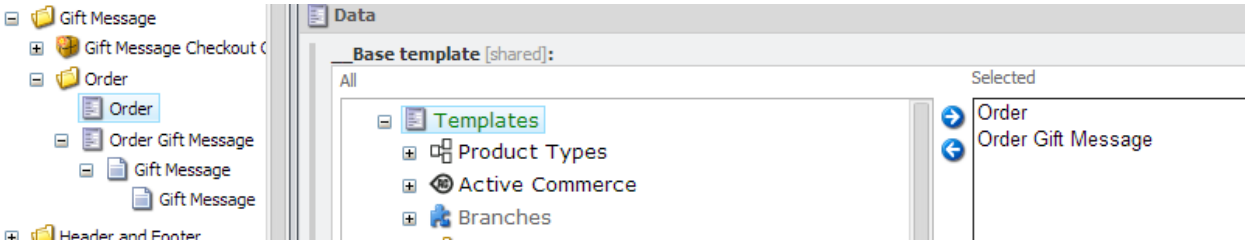
We need to add a new endpoint for our front-end service to call.

- 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

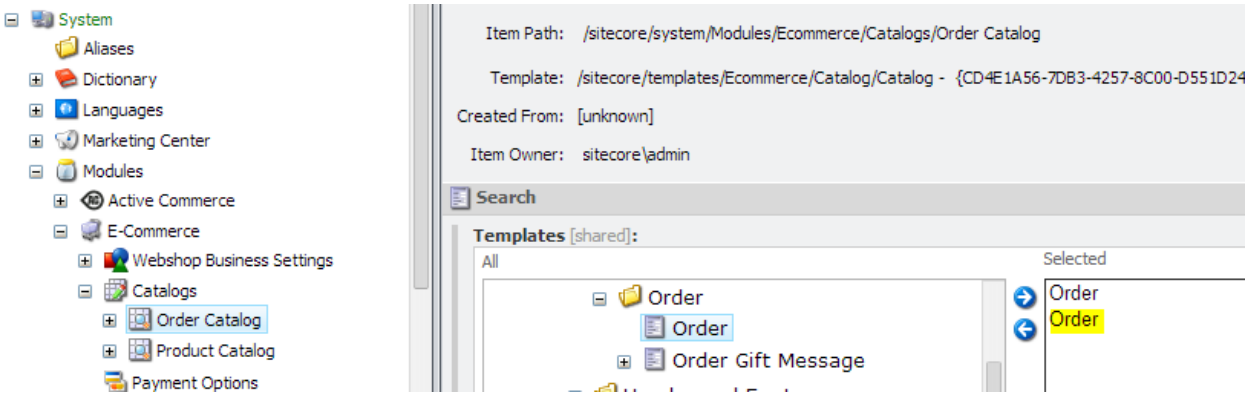
4. Extend the Order

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

- Template - Inherit the base Order template (*/ActiveCommerce/Order/Order*), adding a "Gift Message" field



- Config patch - Inform OrderManager of our new Order template
- Order - Extend the base Order class, adding our "GiftMessage" property
- OrderMappingRule - Extend the base OrderMappingRule, adding on the mapping for our GiftMessage -> "Gift Message" field
- Register in Unity - Use our versions of Order and OrderMappingRule
- Add our new order template to the order catalog "Templates".



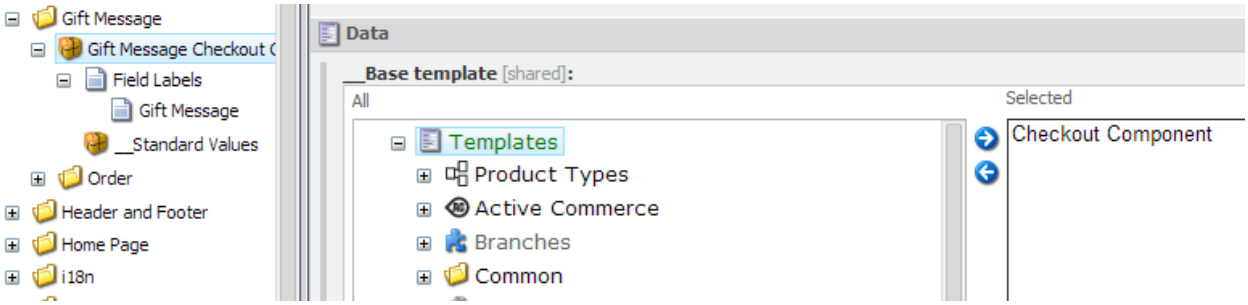
5. Add an OrderPipelineProcessor

- 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 orderProcessing 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.

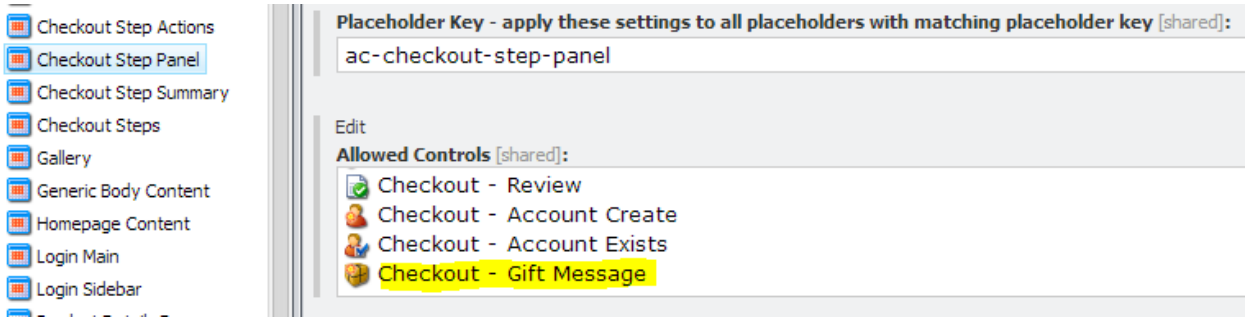
- Template - Inherit the base Checkout Component template (*/ActiveCommerce/Purchasing/CheckoutComponents/Checkout Component*), adding a "Gift Message" label field

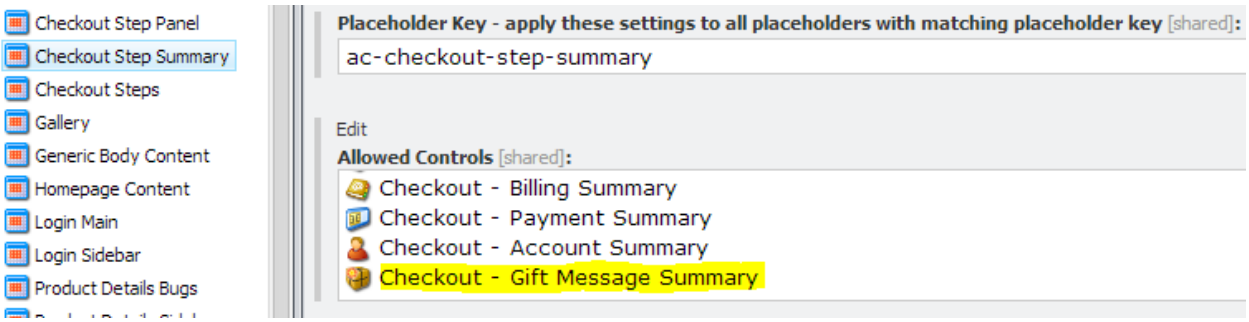


- GiftMessageCheckoutComponent - Create mapped concrete class for our new template (using Glass Mapper)
- View + Code Behind - Contains gift message form to present to customer
- Summary View + Code Behind - Contains summary view of gift message component
- Sublayout, Summary Sublayout - Corresponding 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)